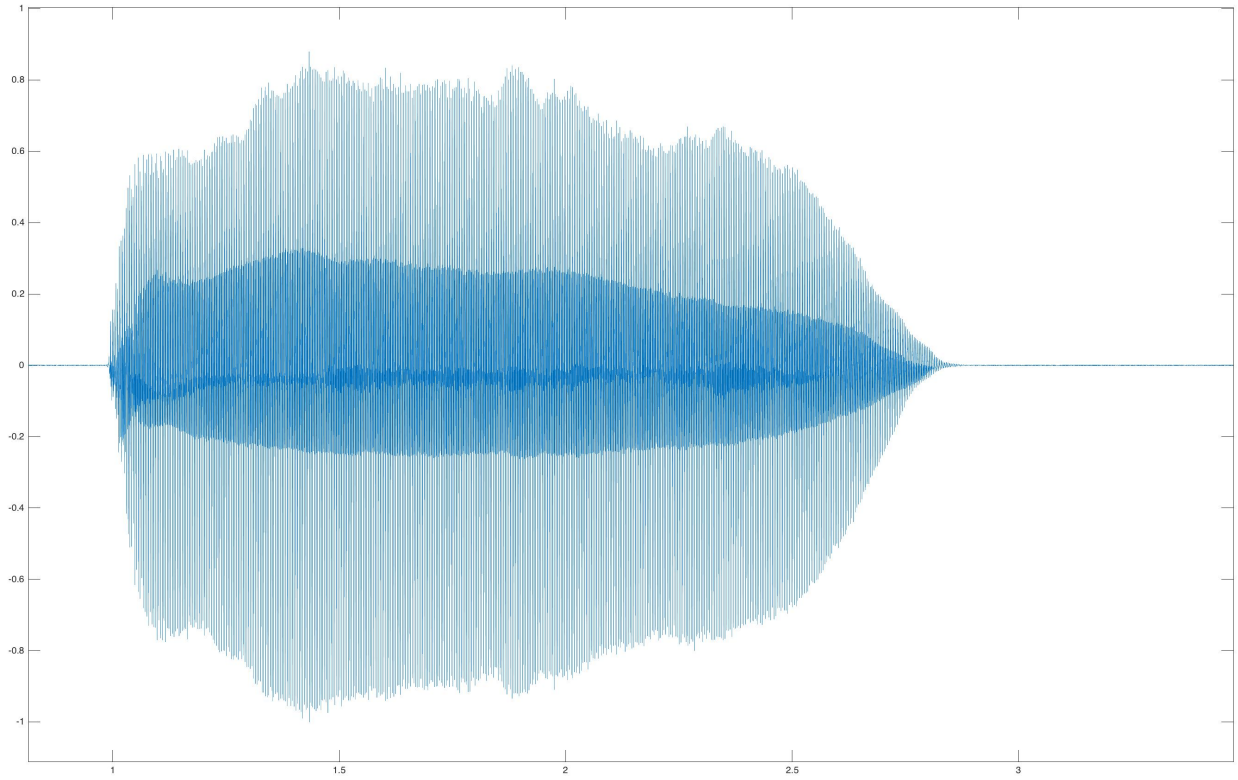


TROMBONE SYNTHESIS USING DEEP LEARNING

A Degree Thesis submitted to the Faculty of the Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politéctinca de Catalunya

By Víctor Badenas Crespo

In partial fulfillment of the requirements for the degree in Telecommunications Systems Engineering

Advisor: Antonio Bonafonte

Barcelona 2017

ABSTRACT

In this project we present a possible expansion of the sampleRNN project for trombone synthesis. The project is divided in three main blocks: the database creation, the sampleRNN analysis and the sampleRNN modification.

The first block contains the generation of a database suitable for the project's application in the second and third block.

The second block consists on an analysis of the published code of sampleRNN and a first approach on training the model with a portion of the database.

The third and final block explains the modifications made to the code to be able to accept note and length values in the input vector of the Neural Network.

Unfortunately, due to time constraints, the expected results were not obtained, as explained in the results and conclusion of the document.

RESUM

En aquest projecte presentem una possible ampliació del projecte anomenat sampleRNN per utilitzar-lo per la síntesi de ones d'audio de Trombó. El projecte esta dividit en tres fases: la creació d'una base de dades, l'analisi de sampleRNN i la modificació del codi.

El primer bloc conté la generació de una base de dades adaptada per la aplicació desitjada en el segon i tercer bloc.

El segon bloc constisteix en l'analisi del codi publicat de sampleRNN I un primer entrenament del model amb una fracció de la base de dades.

I el tercer i bloc final explica les modificacions que s'han fet al codi perquè aquest pugui acceptar valors de la nota i de la llargada de la nota com a entrada a la Xarxa Neuronal.

Malauradament, degut a una manca de temps, els resultats no han estat obtingut tal i com s'explica a l'apartat dels resultats i la conclusió del document.

RESUMEN

En este proyecto se presenta una posible ampliación del proyecto sampleRNN para la síntesis de ondas de audio de Trombón. El proyecto esta fragmentado en tres fases: la creación de una base de datos, el análisis del código de sampleRNN y la modificación del mismo.

El primer bloque contiene la generación de una base de datos adaptada para la aplicación deseada en el segundo y tercer bloque.

El segundo bloque consiste en el análisis del código publicado de sampleRNN y un primer entrenamiento del model con una fracción de la base de datos.

Y el tercer bloque y bloque final explica las modificaciones llevadas a cabo en el código para que sea capaz de aceptar los valores de nota y duración como entrada a la Red Neuronal.

Desafortunadamente, debido a la falta de tiempo, los resultados deseados no se han obtenido tal y como se explica en los apartados conclusión y resultado del documento.

DEDICATION

To my family and their countless hours supporting me through the career, either emotionally or economically and for always keeping my hopes up even when I was having a breakdown.

To Clara for always having my back, and even though she did not understand a word of what I was doing, she always supported me.

To my friends in Angangas for helping me find the passion in music and in what I do.

ACKNOWLEDGEMENTS

First and most importantly, I would like to thank my advisor Antonio Bonafonte for its invaluable help in the development of this project.

I would also like to thank Santiago Pascual for its help in learning the Pytorch environment.

I would also like to thank the Veu group in the Signal Theory and Communications Department for allowing me to use its computing power.

REVISION HISTORY AND APPROVAL RECORD

Revision	Date	Purpose
0	15/09/2017	Document creation
1	27/09/2017	Document revision
2	29/09/2017	Document revision
3	01/10/2017	Document revision
4	04/10/2017	Document revision
5	09/10/2017	Document Approval

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Victor Badenas Crespo	victor.badenas@gmail.com
Antonio Bonafonte Cavez	antonio.bonafonte@upc.edu

Written by		Reviewed and approved by	
Date	27/09/2017	Date	09/10/2017
Name	Victor Badenas Crespo	Name	Antonio Bonafonte Cavez
Position	Project Author	Position	Project Supervisor

TABLE OF CONTENTS

LIST OF FIGURES

1.INTRODUCTION

Audio generation is a complex task in it's whole. The high requirement in computation when implemented in real-time is a struggle to programmers and overall, to everyone working in the field. The evolution of audio synthesis has experienced a huge improvement in the last decade, as hardware became powerful enough to support the needs of the field.

As the usual sample rate in audio is 44,1kHz and 16 bit depth, most environments which work in the field are used to work at higher sample rates in order to avoid aliasing in mixing, mastering and digital synthesis as well as working at higher bit rates to avoid quality loss. Those sampling frequencies are usually 48kHz, 96kHz, 192kHz, and 24 or 32 bit depth, which demand an extremely high computation to operate correctly.

In the music industry, digital synthesizers of many kinds are mainly used for music creation in many environments such as Electronic Music Production, Film Scores and different layering mixing techniques. However, the synths used for those applications are wavetable or additive synthesis, but with the recent success of Deep Learning in a lot of fields, the interest in Deep Leaning based synths has risen. Thanks to computing power greatly enhanced by Graphics Processing Units (GPU), a level of computing power has been reached so that it is possible to work with Deep Neural Networks with reasonably short training times when mapping the speech features from the source speaker to those of the target speaker.

This chapter contains a brief introduction to Audio Synthesis and Deep Learning history and the main constrains that could potentially have had an impact in the development performed. It also provides an overview of the current state of the project and the requirements that the different components of project have to meet.

1.1 OBJECTIVE

The objective of this project is to develop a digital synthesizer of an analog instrument's waveform (trombone) controlled by a midi pattern. The result of the project should be an audio signal with a recognizable pitch and timbre. To do that, the sampleRNN[1] project is used, which is implemented in Pytorch.

[1] paper regarding sampleRNN https://arxiv.org/pdf/1612.07837.pdf and the repository with the code: https://github.com/deepsound-project/samplernn-pytorch (the one in the paper is in Theano, which is an earlier implementation of the same network).

Sample RNN is an implementation of a Recursive Neural Network (RNN) Deep Leaning (DL) model that does not accept other variables other than the previous samples of the waveform. Therefore, it generates random unconstrained music by its own. The main contribution of this project is to accept other types of input information streams beside the already implemented previous sample conditioning.

1.2 GOALS OF THE PROJECT

The main goals of this project are to:

1. Using sampleRNN to generate the audio waveform.
2. Modifying sampleRNN to match the specifications of the project.
3. Being able to generate raw audio at a 16kHz sample rate and 8bit depth.
4. Generating single note files.
5. Creating a database to train the system.

1.3 REQUIREMENTS AND SPECIFICATIONS.

To ensure the correct execution of the software:

- The computer must be running Matlab 2015b or newer, as some of the functions used in the implementation are exclusive to those versions.
- The user should be able to generate a midi file using a DAW (Digital Audio Workstation) such as Ableton Live, Presonus Studio One, Avid Pro Tools, Steinberg Cubase, Fruity Loops Studio or, as in the project, Logic Pro X.
- The computer must be running python 3.5 or higher and must install the following libraries: torch, natsort, librosa, numpy, matplotlib, math, os and pickle.
- The user should have access to a dedicated GPU in order to train or execute the software.

The software will be working at 16kHz and 16bit depth, with 4 second (64000 samples) input sound files and will generate 16kHz 8bit audio files.

1.4 BRIEF HISTORY OF SOUND SYNTHESIS

One of the first electric devices to produce a sound was the musical telegraph, based on a single note oscillator. This device was invented in 1876 by accident by Elisha Gray, who also built speakers later on to be able to listen to the electrical signal.

In 1906, Lee De Forest invented the first amplifying vacuum tube. This led to new technologies such as radio and sound films, but it also influenced the music industry and resulted in early musical instruments that used them such as the Theremin.

In the 1930s and 1940s, the basic elements required for the newer form of synthesis were invented: audio oscillators (wavetable oscillators which could generate on cycle of diverse waveforms), audio filters, envelope controllers and various effects. And were used to develop more electronic-heavy synths. It also was the decades in which polyphonic synths were invented (more than one note at the time).

In the late 1940s, Hugh Le Caine invented a voltage-controlled electronic instrument that provided the three main parameters that we know nowadays: volume, pith and timbre, which correspond to today's touch-sensitive keyboard, pitch and modulation controllers.

From then on, different methods of audio synthesis were discovered: Frequency Modulator Synthesis (FM), Additive Synthesis, Subtractive Synthesis basically. Then, when digital synths were available, granular synthesis, Wavetable Synthesis and sample-based synthesis were available to develop and have become a huge part of the market as digital soft-synths became cheaper and way more useful.

1.5 BRIEF HISTORY OF DEEP LEARNING

It's story goes back to the 1940s, but over the past 5 years, Deep Learning has gone from a somewhat little field of a cloistered group of researchers to being a worldwide mainstream phenomenon. Interest in Deep Learning has sky-rocketed, with constant coverage in the popular media such as top journals like Science, Nature Methods and JAMA among others. DL has learned to drive a car, diagnosed skin cancer and autism and can even create photorealistic pictures. As a good example of the growth in DL interest, the NIPS's (Neural Information Processing Systems) conference has experienced a lot of new methodological research papers published in the last 5 years, as shown in the figure (1).

1.5.1 1940

Early work in machine learning was largely informed by the current working theories of the brain. The first investigators to explore the area were Walter Pitts and Warren McCulloch. They had developed a technique known as "thresholded logic unit" and was designed to mimic the way a neuron was thought to work (which will be a recurring theme). But it isn't until Frank Rosenblatt's "perception" that we see the first real precursor to modern neural networks. For its day, this thing was pretty impressive and it came with a learning procedure that would probably converge to the correct solution and could recognize letters and numbers

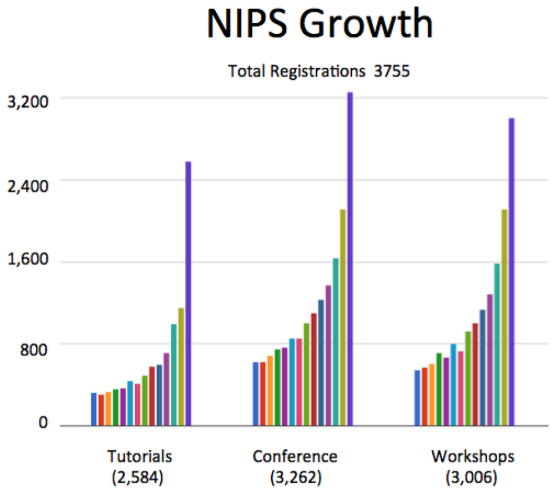


Figure 1: from [4]

1.5.2 1969

Along with the double-PhD wielding Seymor Papert, Marvin Minsky wrote a book entitled Perceptrons that effectively killed the perceptron, ending embryonic idea of a neural net. They showed that the perceptron was incapable of learning the simple exclusive-or (XOR) function. Worse, they *proved* that it was theoretically impossible for it to learn such a function, no matter how long you let it train. Now this isn't surprising to us, as the model implied by the perceptron is a linear one and the XOR function is nonlinear, but at the time this was enough to kill all research on neural nets and begin a long period in which no research was done in the field.

1.5.3 1986

Geoff Hinton finished his PhD studying neural networks in 1978 and by 1986, along with David Rumelhart and Ronald Williams, Hinton published a paper: “Learning representations by back-propagating errors”. In this paper they showed that neural nets with many hidden layers could be effectively trained by a relatively simple procedure. This would allow neural nets to get around the weakness of the perceptron because of the additional layers endowed the network with the ability to learn nonlinear functions. Around the same time it was shown that such networks had the ability to learn any function, a result known as the universal approximation theorem[2].

1.5.4 2006

In 2006, the idea of unsupervised pre-training was introduced by Hinton once again. The main idea behind that concept was to train a 2-layer unsupervised model, freeze the parameters, add another layer and just train that layer. Then adding multiple layers to the network until you had a deep network.

Using this strategy, people were able to train networks that were deeper than previous attempts, prompting a rebranding of Neural Networks to Deep Learning.

1.5.5 2010 and forward

With the incorporation of Graphic Processing Units (GPUs) to train models, the accuracy and the speed of training Deep Learning models increased exponentially compared to training with CPU power. GPUs are parallel floating-point calculators with a large quantity of cores. More speed with GPUs meant that larger models could be trained, which meant lower error rates.

In addition to that the method known as dropout was introduced to prevent overfitting and used the Rectified Linear Activation Unit (ReLU). One large example of the improvement described above is the “Alexnet” network.

[2] https://en.wikipedia.org/wiki/Universal_approximation_theorem

1.6 TIME PLAN

The Gantt diagram and the time plan are detailed in the third annex.

Once the history of Deep Learning and the History of Sound Synthesis have been briefly exposed, the technical specifications needed for the development of the project will be explained in 2.State of the Art and further in the document, the Methodology, Results and Conclusions will be exposed.

2.STATE OF THE ART

2.1 DIGITAL AUDIO SYNTHESIS

A Digital Synthesizer is a piece of software which is capable of generating a stream of numbers representing the voltage outputs of each sample of an audio waveform in the digital domain. Instead of using analog electronics and samplers that play back recordings of acoustic, electric or electronic instruments as analog synthesizers do, digital synthesizers use Digital Signal Processing techniques to generate the sound. Some digital synthesizers emulate analog ones.

A Digital synthesizer is in essence a computer with a piano keyboard (or a midi file as an input) and a LCD screen as an interface. Because of the rapid advancing of computational power, it is often possible to offer more features in a digital synthesizer than in an analog one at a given price. For instance, some forms of synthesis as sampling and additive synthesis are not possible regarding an analog synthesizer, but many musicians prefer the character or the warmth of an analog synth over their digital modeled software.

As stated before, a digital synthesizer generates a stream of audio samples, those are audible by sending the samples through a DAC (Digital/Analog Converter) which will convert the stream to a continuous voltage that will be amplified and reproduced by a speaker. The main concern is to generate those samples, and to do that, one of the most efficient way to obtain it is to get a list of values, called a wavetable which contains periodic information of the wave to generate regardless of the pitch (frequency) at which the synth is generating the signal. To repeatedly scan a wavetable is called table-lookup synthesis and, as computers take somewhat similar to a nanosecond to read a value from memory, table-lookup synthesis is an efficient way of modeling an analog oscillator. The name given to the block which performs table-lookup synthesis is called digital oscillator.

2.1.1 Wavetable Synthesis

Wavetable Synthesis employs the use of a table with various switchable frequencies played in certain orders. The sound moves in order through the wavetable, smoothly changing its shape into the various waves in the table.

This method produces sounds that can evolve really quickly and smoothly. The method was intended to create digital sounding noises, so it is not used for instrument replication very

often, but is an effective way to create pads or harsh-sounding tones like bells or digital sounds.

Some examples of commercial Wavetable Synthesizers are Xfer Records Serum, Lennar Digital Sylenth1, Native Instruments Massive, etc.

Figure 2: example of a wavetable

2.1.2 Subtractive Synthesis

This is the most common method that gave birth to the concept of sound-synthesis.

Subtractive Synthesis consists of simple signal chain regarding an oscillator running going into an EQ filter sent to an amplifier for gain staging.

The main principle behind Subtractive Synthesis is that any harmonic character can be constructed by an oscillator, or the combination of multiple oscillators. Then, by running these oscillators through various filters, and controlling the envelope response (an amplitude modifier), the harmonics can be represented as harmonic structures that mirror those of actual instruments.

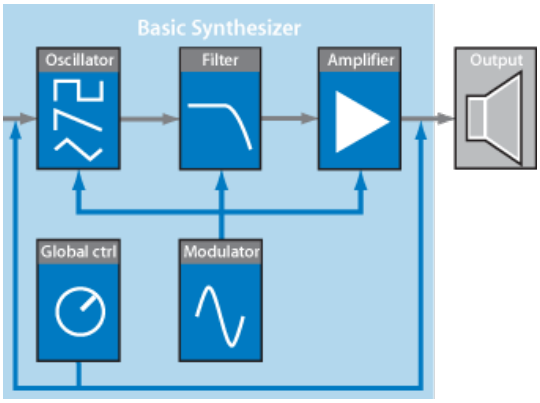


figure3: block diagram of a subtractive synth

2.1.3 Additive Synthesis

Additive Synthesis is trying to achieve the same result as Subtractive Synthesis, but approaches the method creating frequencies instead of filtering them. Rather than generating an spectrum and filtering out the harmonic structure desired, in additive synthesis multiple sine waves of varying levels and frequencies are combined together to build the harmonic structure desired. Equation 4.

$$y(n) = \sum_{i=1}^{N} A_i * sin(2\pi * f_i n) \quad (4)$$

2.1.4 Samplers

Sample-Based Synthesis is different from other forms of synthesis because it does not employ the use of oscillators. In their place, recorded samples are the sound source. Each sample is pitch-shifted to span about 5 notes until a new sample is needed (to avoid noticeable distortion).

This method is meant to emulate real instruments by recalling actual samples of those instruments. These types of synthesizers can take up a lot of processing power due to the storage and instant recall of samples.

Figure 5: Native Instrument's Kontakt, a sample based Synth

2.1.5 Deep Learning based synthesizers

Deep Learning synthesis is somewhat recent and it has not been yet fully developed commercially except in the field of speech synthesis where multiple algorithms and software have been released.

With the creation of big data and it's massive datasets, music synthesis using this method has become more feasible, because of that, multiple projects have been emerging regarding

this field such as NSynth (implemented by google in Magenta, Tensorflow), Wavenet (also developed by google in Tensorflow) or sampleRNN, the one that will be using in this project.

Deep Learning synthesis also relies in samples but for training the model instead of directly recalling a sample. Because of that and the high computational cost that these synthesizers need, they have not yet overcome sample-based synthesis, as most of DL based synths rely on the previous samples to predict the next one and that is an extremely high speed requirement as they need 44100 samples a second minimum to match the quality of other synths.

2.2 DEEP LEARNING

Deep Learning has become a well known resource in many fields such as Image Processing, Speech Processing and Computer Vision. Its main feature is to be able to learn complex non-linear mapping functions.

There is not such thing as a unique definition of Deep Learning but in general we could say that Deep Learning is a group of automatized learning algorithms. Beyond this common definition, it has features such as:

- Transforming and extracting variables using a concatenation of non-linear processing layers while using as input of each layer the output from the previous one. The algorithms can use supervised learning (when you train with a known output value for each input value) or unsupervised learning (no output).
- They are able to learn multiple level features or data representation. Higher level features derive into low level features to create a hierarchy.

2.2.1 Supervised Learning

Supervised Learning is going to be the training method for sampleRNN. Supervised Learning is a technique used to deduce a given function from the training data. The training data consist on pairs of vectors, the input arguments and the desired results. The output vector can be either a numeric value (regression) or a class label (classification). The main objective in supervised learning is to create a function able to predict the output regardless of the input vector (whenever is valid) after seeing a series of examples (training dataset). To

do that the model has to be able to generalize into a family of objects to foresee data that has not seen before and give the correct output.

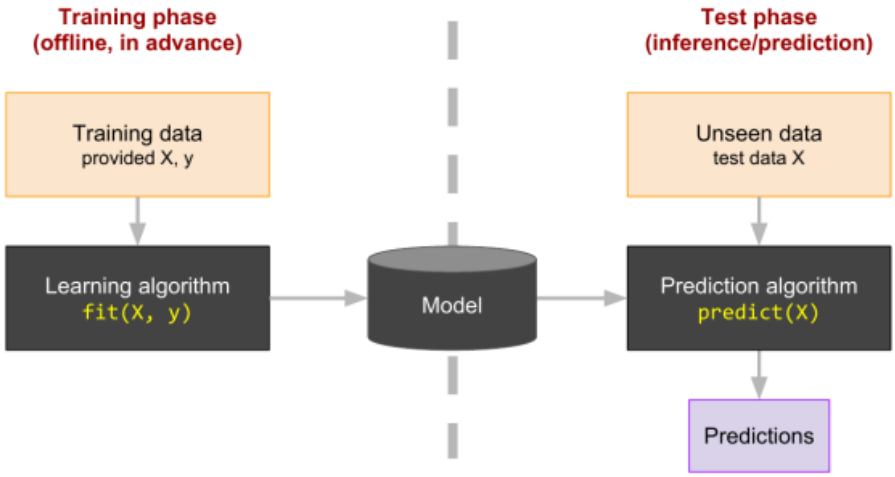


figure 6: diagram of a supervised learning model from [12]

2.2.2 Linear Model

Also known as the most basic Neural Network. It was implemented in the Perceptron, one of the first models using what later on will be known as Deep Learning. It consists on a simple diagram which was named single neuron model for it's simplicity and similarity to a real neuron.

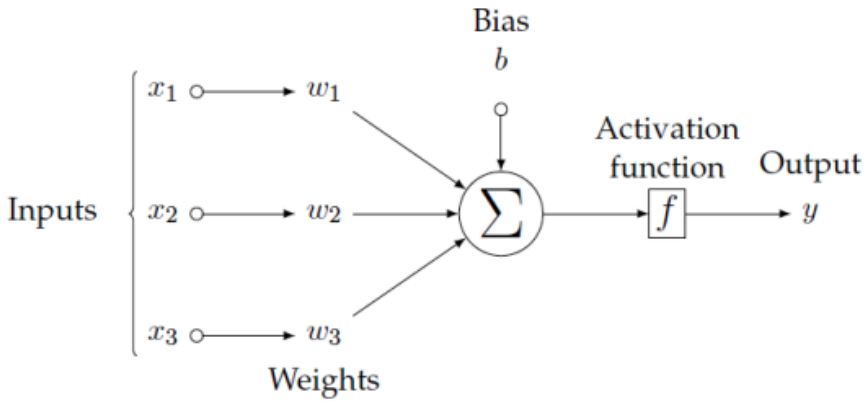


figure 7: diagram of a single neuron model from [12]

The diagram shows a graphical representation of the model, which can also be written and represented in matrix form (Equation 8).

$$Y = f(\begin{pmatrix} \omega_{11} & \cdots & \omega_{1N} \\ \vdots & \ddots & \vdots \\ \omega_{N1} & \cdots & \omega_{NN} \end{pmatrix} X + \begin{pmatrix} b_1 \\ \vdots \\ b_M \end{pmatrix}) \quad (8)$$

Where ω are the weights, b are the bias values and compared to the real neuron model, the weights are the strength of the connection between two neurons and the bias determines how input to a neuron is translated into the state of that neuron.

This model belongs into a category known as fully connected layer in which each output depends on all input values, with every weight ω is different than 0.

The function f is the activation function, which is a non linear function that is mainly used to connect one neuron after another because without the non-linear function would resolve in:

$$Y = W_2(W_1X + b_1) + b_2 = W_2W_1X + W_2b_1 + b_2 = W'X + b' \quad (9)$$

Which is a different linear function and thus, it is not able to learn non-linear regressions and, more importantly, it causes no improvement in relation to a model with only one layer. But, if a non-linear function is added to both layers it takes a step further in complexity (Figure 10)

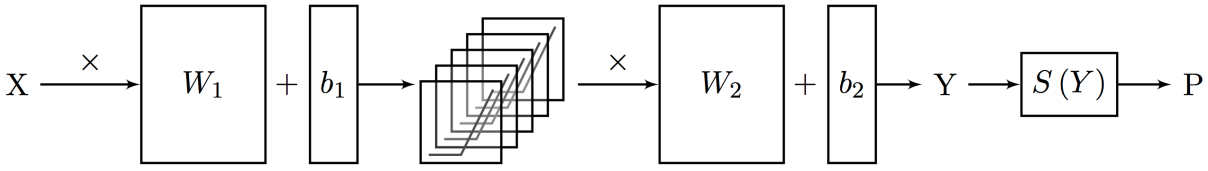


Figure 10: Block Representation of 2 layers (from [41])

2.2.3 Activation function

As mentioned above, the activation function is a non-linear function between layers than allows the neural network to be concatenated without becoming another linear regression. The most common activation functions are the following the sigmoid function (Equation 11.1), the ReLu function (Equation 11.2), the Softmax function (Equation 11.4) and the hyperbolic tangent (Equation 11.3).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (11.1) \qquad ReLU(x) = \max(0, x) \quad (11.2)$$

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \quad (11.3) \qquad S(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, z \in \mathbb{R}^K, j = 1, \dots, K \quad (11.4)$$

2.2.4 Training the model

The model is trained updating their parameters according to a loss function such as the Mean Square Error (MSE) or the Cross Entropy. The result of this function is an indication of the prediction errors from the model.

In order to adjust the weights of the DNN, we use an Optimizer. The most basic of them and the first used in these algorithms is the Gradient Descent. This method updates the weights (Equation 12) by defining a learning rate η which defines how quick the method converges and subtracts the derivative of the loss function in terms of the weights in order to obtain the value of ω for which the error loss is minimum.

$$\omega_{n+1} = \omega_n - \eta \frac{\partial \mathcal{L}}{\partial \omega} \quad (12)$$

The biggest issue in this method is that as the Loss function has a very complex dependance on the coefficients, Gradient Descent does not ensure that the minimum value of the loss function is reached, for that matter, it is possible to never reach the minimum.

However, the speed at which the Gradient Descent function converges is very slow and to solve that, optimizers were implemented, which modified the learning rate and the direction with different strategies. The one used in this project is the Adam Optimizer.

Some networks never learn with enough accuracy to be usable with new data. This could be because the input data do not contain the specific information from which the desired output is derived. Ideally, there should be enough data so that part of the data can be held back as a validation set.

2.2.4.1 Adam Optimizer

Adaptive Moment Estimation (Adam) computes adaptive learning rates for each parameter. It stores the exponentially decaying average of past squared gradients and keeps an exponentially decaying average of past gradients.

It works estimating the first and second order momentum (the mean and the variance) of the gradients, updating them with other parameters that work as a numerical modification speed for those parameters.

2.2.5 Recurrent Neural Networks (RNN)

Recurrent Neural Networks are networks with some fully connected layers and some layers that have shared weights In time. That allows the hierarchy to have memory and thus being able to reduce the number of inputs compared to the same application with a non RNN network. If a desired output value has correlation to an input value located 32 samples before the predicted output, the input values' input window should be at least of 32, but as RNN have hidden memory layers, it is possible to reduce that input window's size while preserving the information of the correlated sample.

The figure on the side represents an RNN model which has the shared weights in the red rectangle.

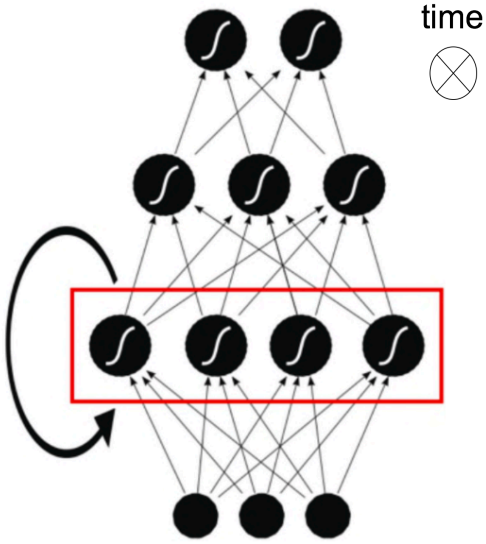


Figure 13.1: Forward in Layers representation of an RNN from [12]

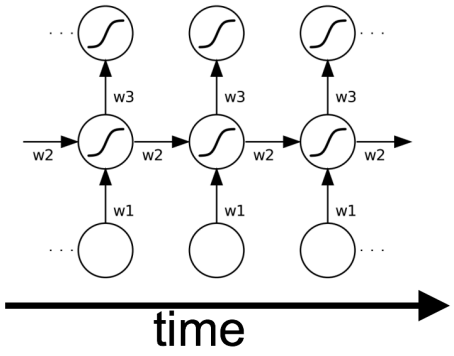


Figure 13.2: Forward in Time representation of an RNN from [12]

$$h_t = f(W \cdot x_t + U \cdot h_{t-1} + b) \quad (14)$$

In Equation 14 it is shown a Time progression of the hidden layer of shared weights.

The figure above represents the update function of the shared weights. W is the weights matrix, x_t is the inputs at the time step (a group of samples at a fixed discrete time), U is the matrix that updates the hidden layer's values for the next time step and b is the bias.

The final equations that will express the answer in terms of the previous hidden state are represented in the diagram if the U matrix is the update matrix and the update function used is an hyperbolic tangent.

$$\begin{cases} h_t = \tanh\left(W_{xh} \cdot x_t + U_{hh} \cdot h_{t-1} + b_h\right) \\ y_t = V(h_t) = W_{hy} \cdot h_t + b_y \end{cases} \quad (15.1)$$

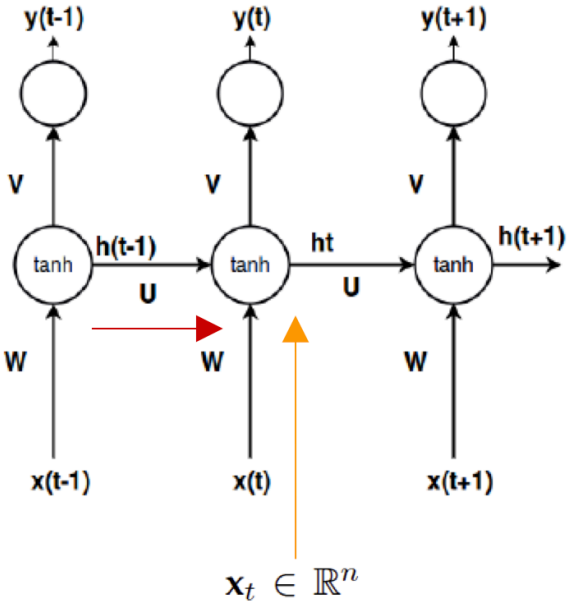


Figure 15.2: example of an RNN from [12]

The main issue with RNN is the vanishing gradient, which is a flaw of RNN cells in which the gradient approaches at early steps as a consequence of multiplying a relatively long sequence of numbers smaller than one, converging the product to 0. This can be an issue when training models with a log sequence of correlated data, as it is the case in Audio.

2.2.6 Long Short Term Memory Units

LSTM is an added complexity to the RNN model, for example in (15.1). It is now widely used because of its revolutionary solution to long term dependences of the sample generated y_t to the previous input samples that are out of reach of the window used in t iteration of the network. LSTM includes gate modules that, unlike all other modules, should be analog for the best performance possible (because of analog's possibility to be differentiable) however, that is not possible so it is instead implemented as element-wise multiplication y sigmoids, which range from 0 to 1.

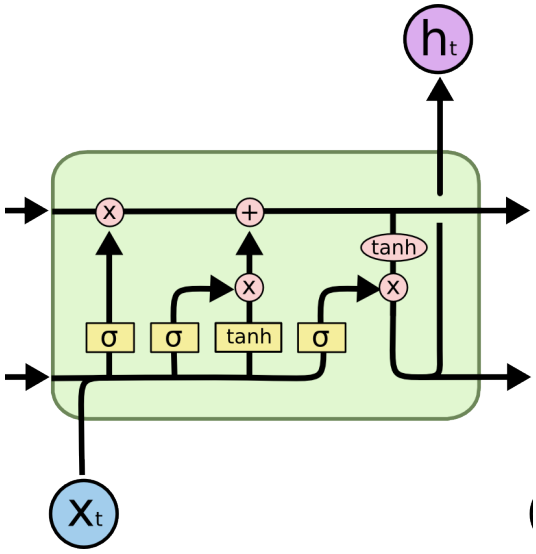


Figure 16.1: block diagram of an LSTM Unit from [34]

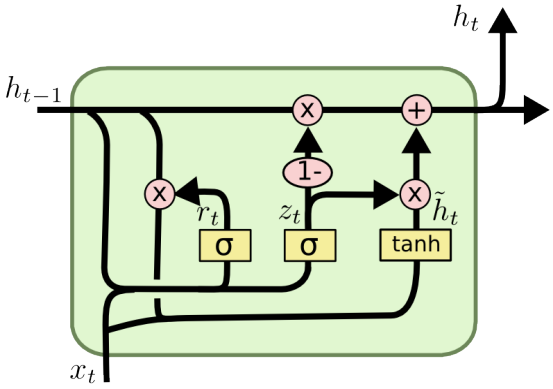
$$C_t = \sigma(W_f * [h_{t-1}, x_t] + b_f) * C_{t-1} + \sigma(W_i * [h_{t-1}, x_t] + b_i) * tanh(W_c * [h_{t-1}, x_t] + b_c) \quad (16.2)$$

$$h_t = \sigma(W_o * [t_{t-1}, x_t] + b_o) * tanh(C_t) \quad (16.3)$$

In the mathematical expression of the LSTM we can see that it consists basically of 4 linear regressions each of them actuating in a different way, and thus, increases the complexity of the network. Each matrix and each bias is trained independently.

2.2.7 Gated Recurrent Units

GRUs or Gated Recurrent Units were developed by Yoshua Bengio in 2014 in his paper [36] and they are a variation of RNN cells. They are easier to train and avoid the vanishing gradient issue that experiment most of the RNN cells. GRUs are less computationally expensive compared to LSTMs and they work as shown in figure(17)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 17: GRU's implementation from [34]

2.3 MIDI

The MIDI protocol or Musical Instrument Digital Interface is a technological standard that describes a protocol, a digital interface and connectors that allow different electronic musical instruments (such as synths), computers and other devices to communicate between them. A simple MIDI connection can transmit up to 16 information channels that can be connected to different devices.

MIDI data carries messages that specify musical notation, tone and velocity among other parameters but, for our application, only the pitch is relevant. Each MIDI file also contains information about the instrument of the General MIDI sound bank used for each channel

and information regarding the Tempo in ms, allowing it to send a master clock track to all the devices connected with the protocol.

These data can also be recorded in a computer in a standalone software called Sequencer, which is a built-it feature in all DAW (Digital Audio Workstation) such as Ableton Live, Logic Pro or Avid Pro Tools which has the capability of saving MIDI files and play it with different sounds afterwards.

Some of the advantages of using MIDI are it's size (as it is not sound but information, a whole song can be stored in a few kilobytes of memory) and the ease of use (modification and instrument selection).

For more information regarding MIDI messages [16].

MIDI was patented by the Yamaha corporation in 2001: [17]

2.3.1 Messages

A MIDI message is an instruction that controls some parameter of the receiver device. It consists on a status byte, which states which type of message is the one that follows it followed y two bytes that contain the parameters. MIDI Messages can be classified as Channel Messages which are sent to some of the 16 channels or as System Messages, which can be heard by all connected devices. Any data non relevant for a device is ignored.

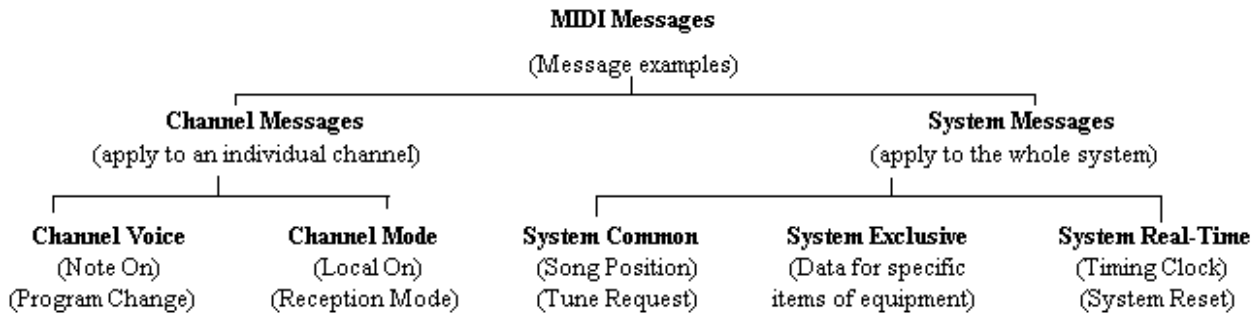


Figure 18: different MIDI messages types

Channel Voice messages transmit real-time performance data through a single channel. Some examples of it are the Note On message, that contain the midi note number that

specifies its pitch and a velocity value that determines the intensity with which the note has been played and the Note Off message that indicates the culmination of a single MIDI note.

Inside the Channel Voice messages, are also included those that change the program and change the devices patch and those messages that adjust or modify instruments' parameters.

Channel Mode messages include the Omni/mono/poly mode on/off messages as well as messages that reset all controllers to its initial state or even to send Note Off messages for all the notes.

2.3.2 System Exclusive Messages

These kind of messages are the reason for MIDI's flexibility and longevity. These messages allow manufacturers to create specific messages for their devices that otherwise, MIDI information would not be enough.

Each Manufacturer has a unique ID that is included in the SysEx messages, which help the messages to only be heard by those devices for whom the message is directed to and ignored by the other ones.

2.4 WAV AUDIO

The Wave Audio Format or WAVE (WAV) is a wrapper that allows to store audio files with different formats.It was developed by Microsoft and IBM and it is commonly used to store high quality digital audio files.

The most common format used in the wave files is PCM (Pulse Code Modulation) either in 16bit or 8bit, and will be the format used in this project.

3.METHODOLOGY

On this chapter all the stages of the project's development will be exposed as well as the methods used to reach the conclusion. First, the creation of the dataset will be explained, then, how the data has been prepared for the model. In the end, the modifications made to sampleRNN to match the specifications stated at the beginning of this document.

For the development of the project, learning a programming language was required. The candidates were: Python [23] (common to all possibilities); to develop a project's exclusive model using Keras [22]; to adapt and comprehend Google's Wavenet implemented in Tensorflow [24] and the one that has finally been used that is SampleRNN with Pytorch [25].

In the development stage of the project (regarding the first months) those possibilities were submitted to an extensive evaluation of viability. It was finally decided that SampleRNN and Pytorch were the ones to better fit the needs of the project as well as having some sort of viability regarding the capabilities of the author.

3.1 DATABASE

The goal of the database creation stage is to create a sound-bank suitable for the desired model. As the first try of implementation was Google's wavenet, a database was created with five different musicians playing simple melodies. That database was lost due to hardware issues and because of the inconvenience that would have been for the collaborators to redo the whole dataset, it was decided to simplify it.

The database for sampleRNN was created with two different musicians playing onto three different microphones in a professional recording studio. The database consists of 904 single-note wav files at a 16kHz sampling frequency and 16 bit resolution.

3.1.1 Sample Recording

The samples were recorded at Beat Studio BCN, a professionally acoustically treated room which helped capture the trombones' sound without early wall reflections. The samples recorded were from five different frequencies: a F2(174Hz), a G2(196Hz), an A2(220Hz), a Bb2(233Hz) and a C3(261Hz) and had variable time decay from 1 to 3 seconds. For the sake of variety, three microphones with different frequency response curves were used to

capture the sound: a Sennheiser MD421 (Figure 19.3), a Shure SM57 (Figure 19.2), and a Behringer ECM8000 (Figure 19.1).

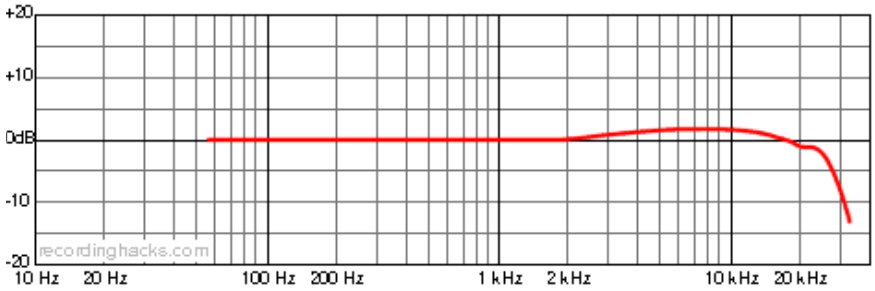


Figure 19.1: Behringer ECM8000 Frequency Response
from http://recordinghacks.com/microphones

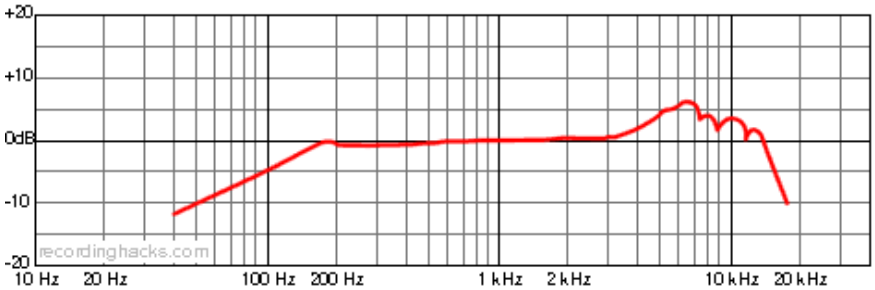


Figure 19.2: Shure SM57 Frequency Response
from http://recordinghacks.com/microphones

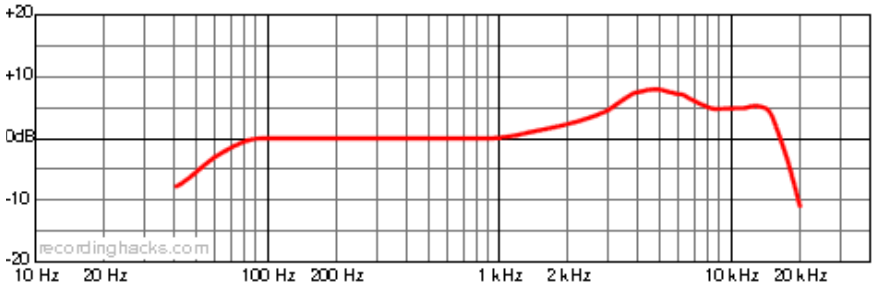


Figure 19.3: Sennheiser MD421 Frequency Response
from http://recordinghacks.com/microphones

After the microphones captured the sound, this was fed with an XLR balanced cable to a MIDAS XL48 microphone preamplifier, which is a transistor-based amplifier with a Low cut frequency of 100Hz. Then the audio signal was fed into an Eventide Orion32, a A/D converter which imported the audio signal to ProTools 10.

Then each note had to be synchronized with a MIDI file for further purposes. To do that, a session of Logic Pro X was used (Figure 20). It has been also used to break the audio files into single-note files.

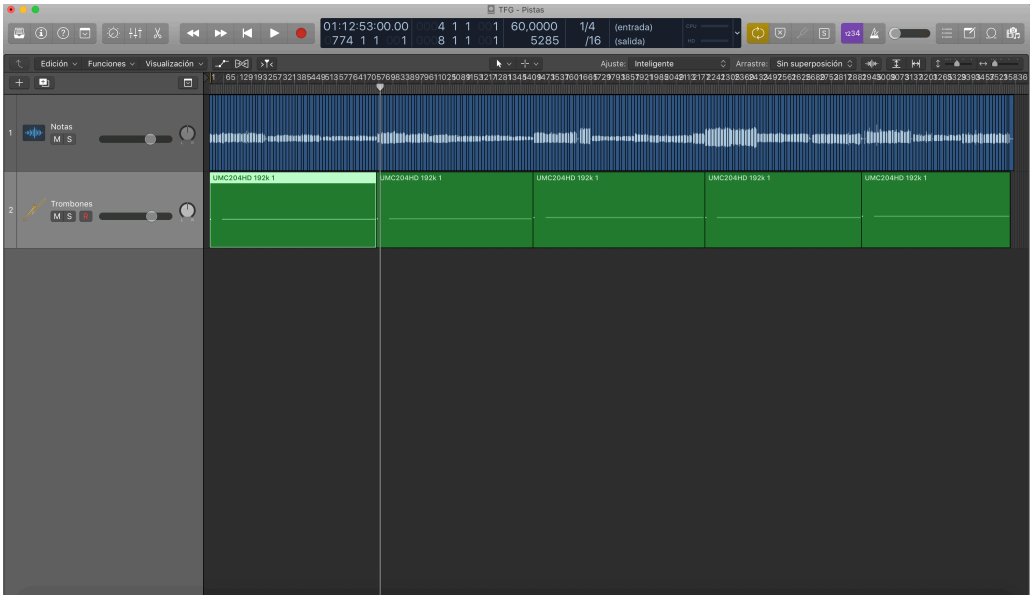


Figure 20: Logic Pro X file

The audio files were then rendered and exported into single audio wav files to further feed the model. But before it is suitable for working as a training sequence, the samples must go under a depuration through MATLAB.

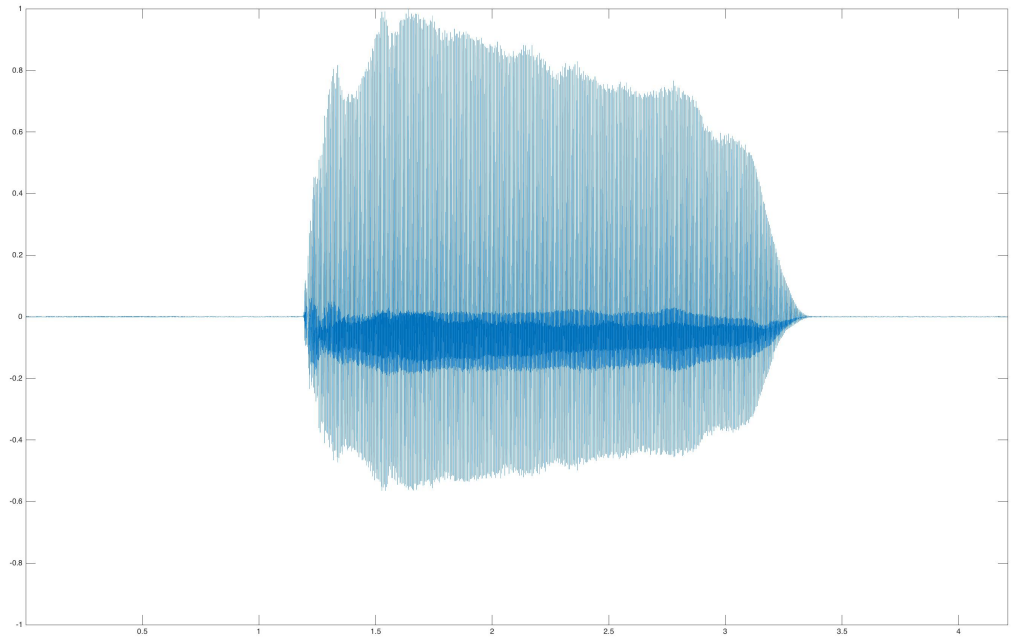


Figure 21: Example audio file (A_50)

3.1.2 Sample Conditioning

This part of the project consists in two main parts, the audio conditioning and the MIDI reading and text file generation.

3.1.2.1 Audio Conditioning

In order to be able to feed the audio to train the model, each wave file must be of the same length, be 8bit resolution wave and have a sampling frequency of 16kHz. To fulfill this need, a Matlab script was created to condition every file all with one script.

To do that, Matlab has to be able to browse through folders, and then regularize the audio samples. The audio files have a length of 176400 samples (4 seconds at a 44,1kHz sampling frequency). It is decided that, to be able to conserve the transients, a 1 second stream of silence will be added at the start of the sample.

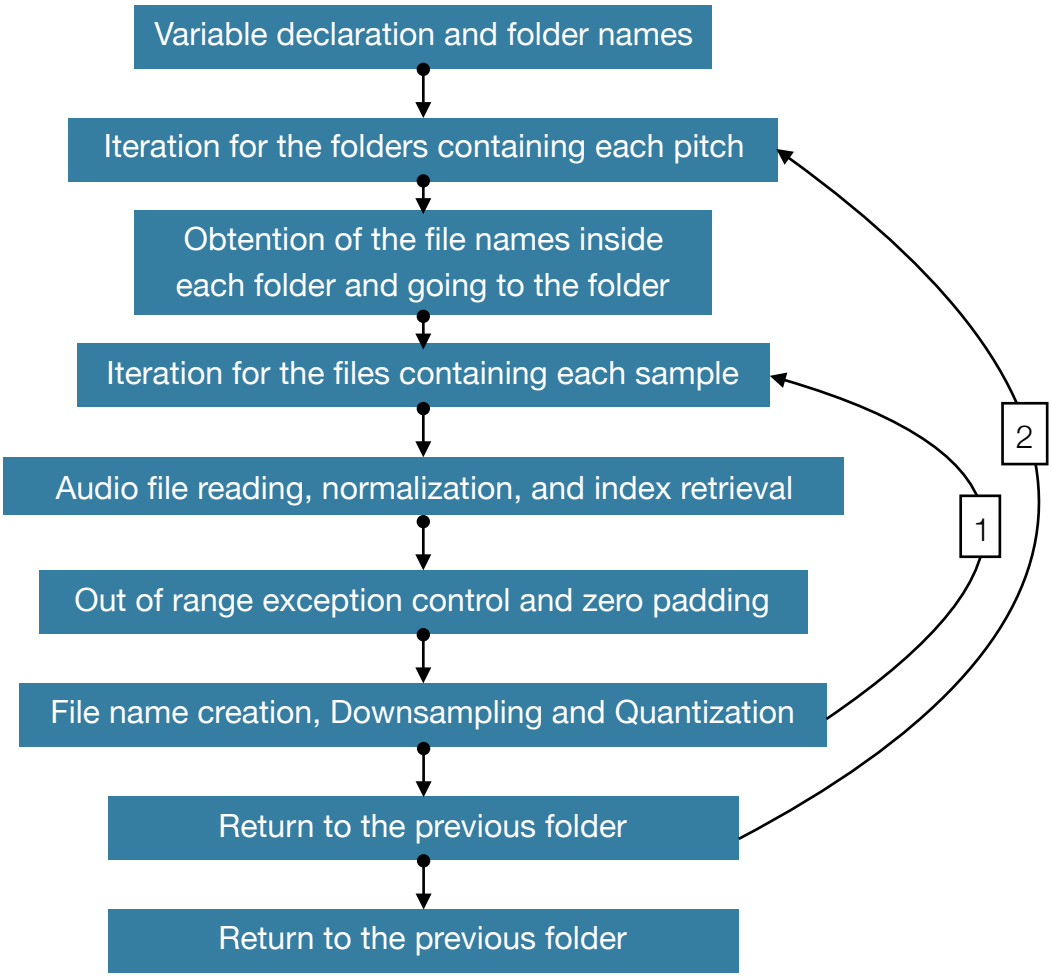


Figure 22: Block diagram of the MATLAB script for audio regularization

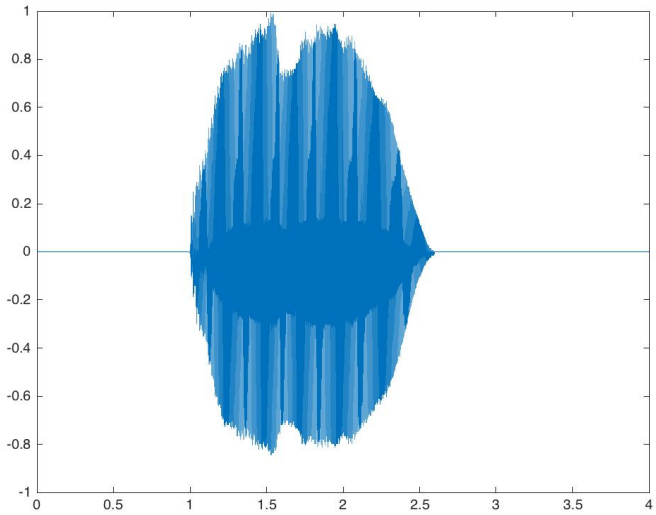


Figure 23: Audio sample after normalization and regularization A_1

The MATLAB Script used to develop this task mainly relies on the dir functionality of MATLAB which allows the program to explore the folders in a MATLAB path. It also relies on the audio reading capabilities of the software which also allows the user to modify the audio file and store it in a new mono wave file.

The signal flow diagram as it is shown in Figure (22) represents the different sections of the script which performs different tasks. 1 and 2 in the diagram represent the iterations in the for loop where the loop 1 iterates for all the files in the directory and 2 iterates for the five folders containing each pitch.

3.1.2.2 Reading MIDI and text file generation

To condition sampleRNN, two different parameters will be used: the MIDI note value and a vector that contains a counter (in samples) until the end of the note.

The MIDI note value is a 7bit coded value (128 values) which correspond to the notes on a keyboard reaching from the -2 octave to the 5th one. In this part of the development of the project, it is used [28] a series of MATLAB/GNU Octave functions that provide the software the capabilities of reading MIDI files and adapting them to be read in a simple matrix among other functionalities.

In the resulting text file, two columns are included. The first one containing the note value for each sample and the second one containing the number of samples remaining until the audio waveform is at the 0.8% of its maximum value. It mainly relies on the previously mentioned functions and on the **fopen** [32] and **fprintf** [31] functions.

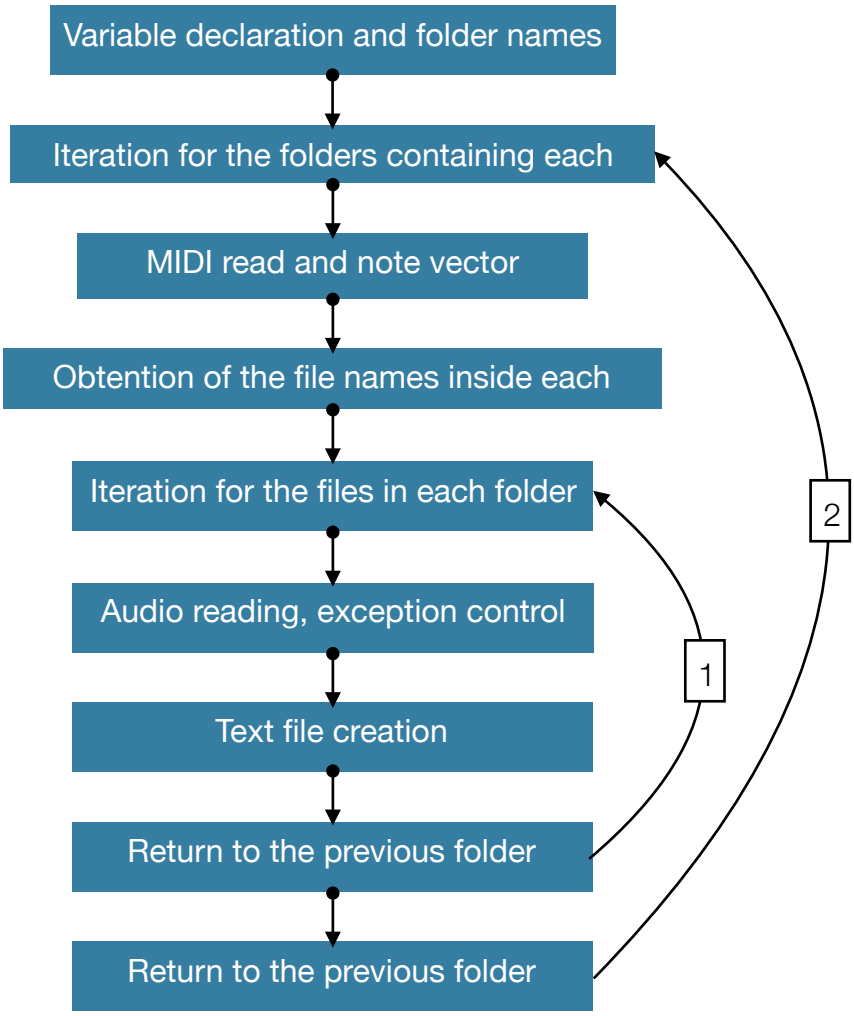


Figure 24: Block diagram for the script to generate the txt variables file

3.2 SAMPLERNN ARCHITECTURE

One of the objectives of the project was to be able to comprehend and analyze how sampleRNN, a previously designed Deep Neural Network, worked. On the theoretical side, the paper attached to this project [33] describes how does the architecture work regardless of the environment in which is implemented (Theano or Pytorch are the ones that have been explored so far, but we are going to use the Pytorch implementation as is the most recent one) and we further analyze the Python implementation for that hierarchy. After successfully doing that, the modifications to accept other input variables will be studied.

SampleRNN is a project that models the probability of a digital audio signal stream $X = [x_0, x_1, x_2, \ldots, x_{N-1}]$ as the product of the probability from each sample conditioned to the ones before that. Figure (25) shows the mathematical expression of the desired expression explained above.

$$p(X) = \prod_{i=0}^{N-2} p(x_{i+1} \mid x_0, x_1, \ldots x_i) \quad (25)$$

This is implemented using a Recursive Neural Network or RNN which, as explained before, is a Deep Neural Network with a hidden state that depends on the previous states and it is also trained. RNNs usually have a function such as shown in Figure (26). Where Υ is one of the known memory cells either a GRU (Gated Recurrent Units) or a LSTM (Long-Short Term Memory Units).

$$h_t = \Upsilon(h_{t-1}, x_t) \quad (26.1)$$

$$p(x_{i+1} \mid x_0, \ldots x_i) = Softmax(MLP(h_t)) \quad (26.2)$$

Because of audio samples being extremely correlated to samples far beyond the actual sample's time step, sampleRNN adopts a multi-model hierarchy in which each model has a different time span in samples, because of that it is able to recognize multiple samples before the actual sample to use them in the prediction of the next one.

The hierarchy consists of two main categories of modules:

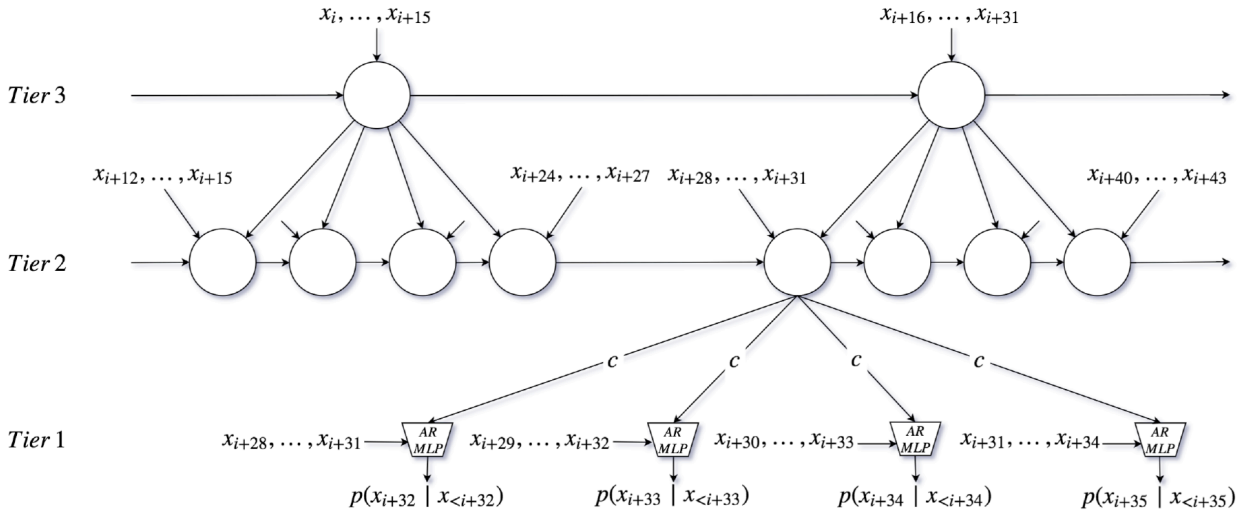


Figure 27: Block diagram of the model for 3 tiers [33]

3.2.1 Frame Level Modules

Frame Level Modules are implemented in the higher tiers of the hierarchy and operate with non overlapping frames of the length frame-size FS^k for the k^{th} tier in the hierarchy. In the case of Figure 28, $FS^3 = 16$ and $FS^2 = 4$. As a regular RNN, it stores a memory from the previous time step and updates it as a function of the previous hidden state $h_{t-1}^{(k)}$ and the input vector $x_t^{(k)}$. As each module operates with a different temporal resolution, an upsampling module is needed. So each vector c is upsampled to $r^{(k)}$ vectors before feeding it to the next module. ($r^{(k)}$ is the ratio between the output of the previous module and the input of the next one)

Figure 28 shows the mathematical expression for a frame level module given a time step t for simplification where $f^{(k)}$ represents the frame input for the k^{th} module.

$$inp_t = \begin{cases} W_x f_t^{(k)} + c_t^{(k+1)} & for\ k < 1 < K \\ f_t^{(k=K)} & for\ k = K \end{cases} \quad \text{(28.1) from [33]}$$

$$h_t = \Upsilon(h_{t-1}, inp_t) \quad \text{(28.2) from [33]}$$

$$c_{(t-1)*r+j}^{(k)} = W_j h_t \quad for\ 1 \le j \le r \quad \text{(28.3) from [33]}$$

3.2.2 Sample Level modules

Sample Level modules are always located in the first tier $k = 1$ of the hierarchy and takes the resized output of tier2 which will notate as $c_i^{(2)}$ and its $FS^{(1)}$ which will contain the preceding samples of the sample x_{i+1} which is the one that the final tier (this one) must predict. To perform this task and given that the correlation between samples that are this close is somewhat small, a Multi Layer Perceptron (MLP) is used instead of an RNN architecture which will speed up the process slightly. The previous samples are quantized to

8bit instead of 16 by a linear quantization function and the resulting samples will be noted as e_i. Figure 29 represents the mathematical expression of the layer which contains a matrix named W_x which purpose is only to match the input from the previous tier to the one in $f^{(1)}$.

The flatten function returns a 1-dimensional vector of the input.

$$f_i^{(1)} = flatten([e_{i-FS^{(1)}+1}, \ldots, e_i]) \quad \text{(29.1) from [33]}$$

$$inp_i^{(1)} = W_x^{(1)} f^{(1)} + c_i^{(2)} \quad \text{(29.2) from [33]}$$

$$p(x_{i+1|} \mid x_i, \ldots, x_1) = Softmax(MLP(inp_i^{(1)})) \quad \text{(29.3) from [33]}$$

3.2.3 Sample RNN code analysis

The previous explanation is a summary of sampleRNN paper's aspects concerning this project, but, when inspecting the code to adapt it to multiple inputs, I noticed that some aspects of the actual implementation of the layers were not explained on the paper, thus, I'll explain some of the features noticed.

In the Pytorch implementation of the sampleRNN project there are 6 python files and a module which are: train.py, dataset.py, model.py, utils.py, optim.py and nn.py. The module is the trainer module which has two files: __init__.py and trainer.py. Each script contains different classes with multiple functionalities on the program, in this section, the most important details from each script.

The program is called with a parse of arguments, the ones with special importance are: --exp : the experiment name to create a folder in the results folder ,—dataset : the folder inside the datasets path containing the dataset, --datasets_path contains the information of the path, --n_rnn : number of RNN layers in each tier ,--val_frac and --test_frac : fraction of the dataset that is used for validation and test respectively. ,--keep_old_checkpoints : keeps the state from previous epochs (must have the same parameters) ,--sample_length : each epoch generates a test output and stores it, this parameter (in samples) sets the length of each audio file.

train.py gets the parser arguments and stores them in the params struct, that is the main container of the global parameters of the code. It also initializes all the modules.

In model.py, where the modules of the Figure 27 implementation are implemented, it is noticeable that in each RNN layer inside the frame-level modules and in the MLP modules.

Each Tier in Figure 27 is composed by a 1D convolution that upsamples a [1,n_frame_samples] size matrix into a [1,dim], then a linear regression is used, then a GRU module is implemented with a [1,dim] input and output, then an iteration of rnn modules is used to implement the layer (always using linear regressions as well) and then, to finish each tier, a 1D convolution as the first one, but now the input and output dimensions are the same.

3.3 USAGE OF THE CODE

Using UPC's calcula remotely, uploading the code and executing the program was possible. As a first execution, a single-note database was used to generate audio files without modifying the original code but only for a little detail which is that the -epoch_limit parameters were not correctly implemented and a type=int specification had to be added. Once it was trained with different parameters (different batch_size, epochs and rnn iterations), it was decided that the best result was obtained for the default batch_size (128) and 3 rnn iterations, but the epoch value depends on the note.

Once the code was working, we proceeded to modify the code to fit the other input parameters.

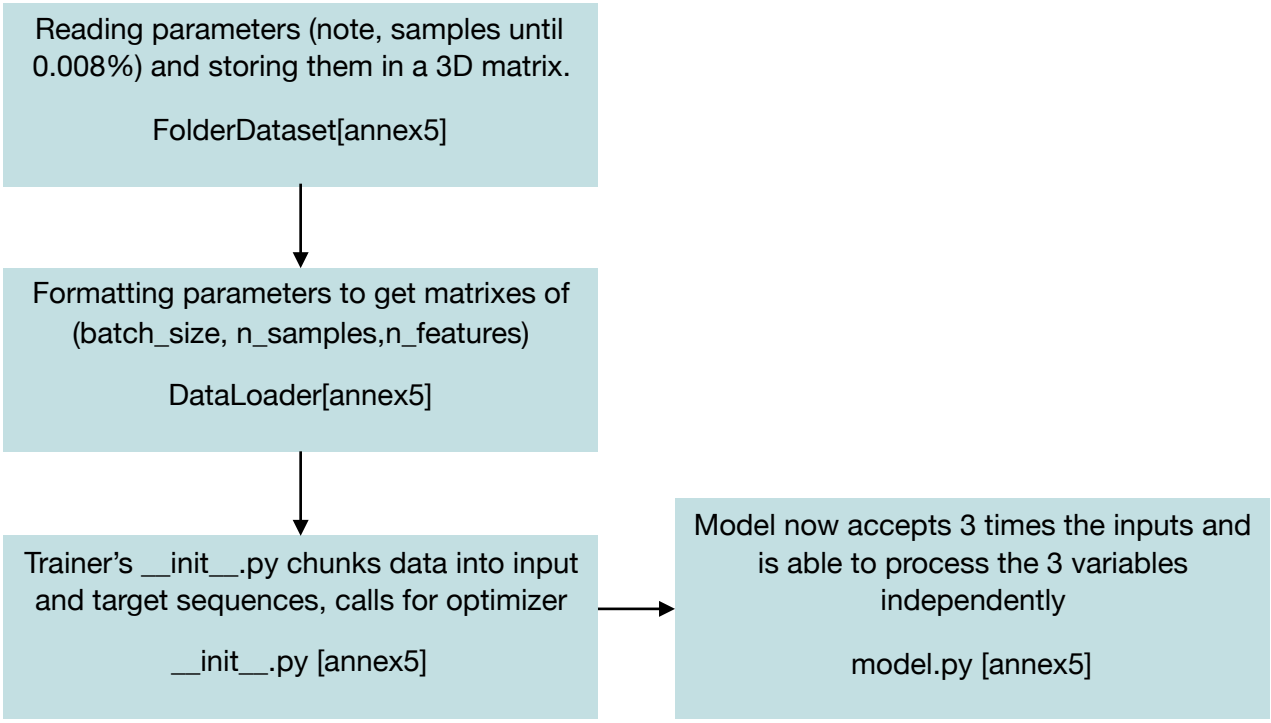


Figure 30: block diagram of the modifications done to the

Once the code for the correct data reading is done and working, there is the need of changing the train function to manipulate the 3D matrixes and there is also a need to get only the audio of the target matrix as the output is going to be still an audio signal and not a 3D matrix. This is done in the __init__ python file of the trainer module.

Then, once the signal arrives in the desired format to the model python file, the Predictor and Generator file must be altered to give the framelevelRNN class the input of 3 times the size in the dimension number 1 to enter the three parameters.

4.RESULTS

As two different phases of the project were developed, the results of each one will be explained accordingly.

On the first stage, as explained before, 5 networks were trained, each one with a trimmed note bunch of the dataset. Because of that 5 different loss functions in terms of the epoch are obtained (each model was trained with 3 rnn and 40 epochs to detect overfitting):

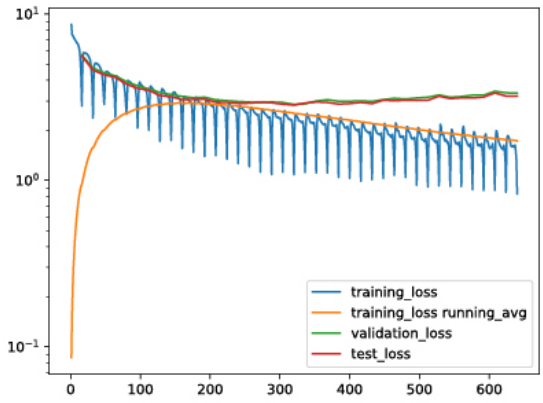


Figure 31.1: TrimF loss diagram

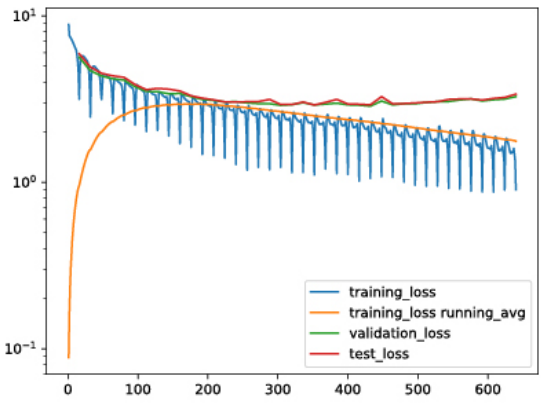


Figure 31.2: TrimG loss diagram

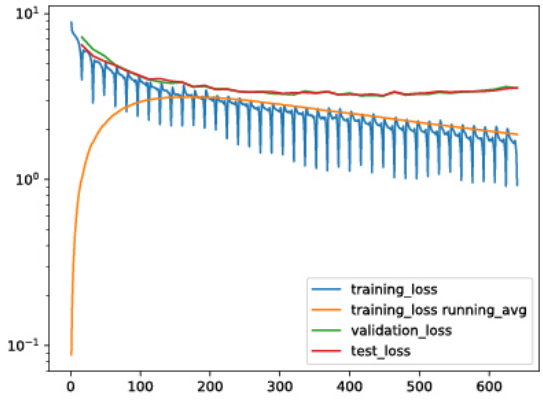


Figure 31.3: TrimA loss diagram

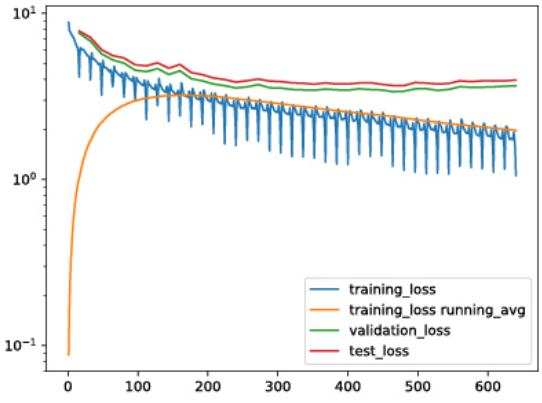


Figure 31.4: TrimBb loss diagram

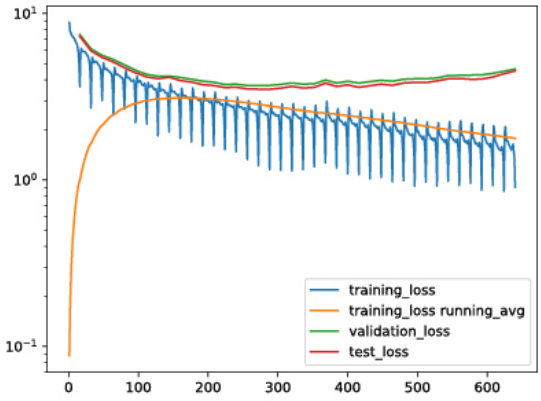


Figure 31.5: TrimC loss diagram

Figures 31.1-31.5 shows the training_loss (blue), training_loss avg (orange), validation_loss (green) and test_loss (red) as a function of the number of files read, where the epochs are:

$$numepoch = files_{read}/files_{inthefolder}$$

The training_loss is the value of the training loss in each train for each batch_size piece of data in the input dataset. Training_loss running_avg is the mean of the training_loss for each epoch. And the validation and training loss graphs are the loss measured with the portion of the dataset used to validate and test the results in each epoch.

On the figures we can clearly see an overfitting of the model by looking at the validation and train loss curves that experiment a growth around the 20th epoch. That means that the best epochs are around the 20 epochs, which generate waveforms such as Figure 32, which is the generated waveform of the note C. Because of the overfitting with a saturated output we can conclude that the model had little samples to train and we are in need of a higher number of files. This should be fixed in the second phase of the project, when the database is three times bigger.

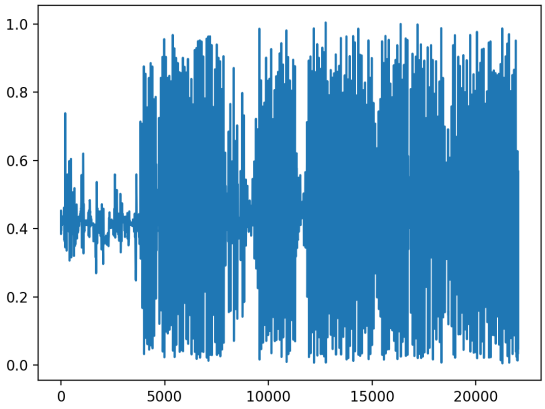


Figure 32: TrimC output signal

For the second phase, due to time constrictions, I was not able to execute it because, even though the model was able to train, It was not capable of generate the audio sample for each epoch because the variables needed for loading the text file and formatting it were not reaching the Generator Class in model.py. For that matter I am not able to present quantitative results for the execution. The only thing to present is the bash output (Annex 4) to demonstrate that the program is training the model and that it prints errors for the Generation Class.

5.BUDGET

For the budget calculations, I contemplated the working hours of myself, for whom I have considered a reduced wage of 10€/h and I contemplated the working hours of my advisor, whom I considered of 60€/h.

The price for the amazon gpu servers were taken from: https://aws.amazon.com/ec2/spot/pricing/ where, as I didn't know what kind was the GPU used in Calcula, I contemplated the most expensive one used with UNIX/Linux for the sake of a "realistic" budget.

The computer's price is taken from a standard average price in laptops, as it does not need a paid OS (because it uses linux) and it only works as a connection to the GPU servers and to run Pycharm.

Descripción	Cantid.	Precio unitario	Importe
Junior Engineer	30h/week	10€/hour	€ 10.500
Senior Engineer	2h/week	60€/hour	€ 4.200
Amazon GPU servers	100h	€ 0,4079	€ 40,79
UNIX Computer	1	€ 500	€ 500
MATLAB License	1	€ 1.000	€ 1.000
Total			**€ 16.240,79**

6.CONCLUSIONS

The project was concluded with the thought that the goal for the project was a little bit too ambitious because of the time needed to finish it. A lot more resources were needed to choose which model to use for that application and that ended up being very time-consuming.

Unfortunately, the desired results were not obtained, even though a partial modification of the code was achieved, and the generation modifications were almost finished.

As for the second stage of the project, the model was successfully trained and generated a sample for each note, but was completely clipping even though the model was overfitted which leads us to think that there were not enough different samples for it to train correctly. That was planned to be fixed once the modifications were done, as the database would be 3 times greater and it would only have 3 times the inputs.

However, the goal in terms of generation was to generate 16kHz 8bit files and, even though the waveform is not quite trombone-like, the pitch of the wave file is fairly recognizable.

7.BIBLIOGRAPHY

[1] Wikipedia (2017,September 15), *Synthesizer* https://en.wikipedia.org/wiki/Synthesizer

[2] Julius Smith (2006, October), *History and Practice of Digital Sound Synthsis* http://www.aes.org/technical/heyser/downloads/AES121heyser-Smith.pdf

[3] Tweakheadz lab, *What are software Synthesizers and the types of Synthesis within today's software instruments* http://tweakheadz.com/software-synths/

[4] Andrew L. Beam (2017, February 23), *Deep Learning 101 - History and Background* https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

[5] Keith D. Foote (2017, February 2), *A Brief History of Deep Learning* http://www.dataversity.net/brief-history-deep-learning/

[6] John Strawn, *Introduction to Digital Sound Synthesis* https://courses.cs.washington.edu/courses/cse490s/11au/Readings/SynthesisChapt3.pdf

[7] Wikipedia (2017,September 15), *Digital Synthesizer* https://en.wikipedia.org/wiki/Digital_synthesizer

[8] Scott Rise, *Wavetable Synthesis* http://synthesizeracademy.com/wavetable-synthesis/

[9] Wikipedia (2017, September 10), *Aprendizaje Profundo* https://es.wikipedia.org/wiki/Aprendizaje_profundo

[10] Jason Brownlee (2016, March 16) *Supervised and Unsupervised Machine Learnu¡ing Algorithms* https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/

[11] (2014, March 13) *Introduction to Computational NeuroScience* https://courses.cs.ut.ee/MTAT.03.291/2014_spring/uploads/Main/Lecture6.pdf

[12] Deep Learning Course Slides by Universitat Politecnica de Catalunya

[13] Rishabh Shukla (2017, January 5), *How to train your Deep Neural Network* http://rishy.github.io/ml/2017/01/05/how-to-train-your-dnn/

[14]*Training an Artificial Neural Network Intro* https://www.solver.com/training-artificial-neural-network-intro

[15] Wikipedia (2017, October 2), *MIDI* https://es.wikipedia.org/wiki/MIDI

[16] *Summary of MIDI Messages* https://www.midi.org/specifications/item/table-1-summary-of-midi-message

[17] Yamaha Corporation, Tomoyuki Kumagai (2001, May 15) *Data sending apparatus and data receiving apparatus communicating data storage control command in MIDI protocol, and method therefor* https://www.google.com/patents/US6232541

[18] https://blog.landr.com/es/que-es-el-midi-la-guia-del-principiante-para-la-herramienta-musical-mas-poderosa/

[19] ¿Que es WAV? https://www.coolutils.com/es/Formats/WAV

[20] (2017, May 31) *Waveform Audio Format (Wav)* https://es.wikipedia.org/wiki/Waveform_Audio_Format

[21] E. Fleischman (1998, June) *WAVE and AVI Codec Registries* https://tools.ietf.org/html/rfc2361

[22] *Keras: The Python Deep Learning library* https://keras.io

[23] *Python:* https://www.python.org

[24] *Tensorflow: An open-source software library for Machine Intelligence* https://www.tensorflow.org

[25] *PyTorch documentation:* http://pytorch.org/docs/master/

[26] https://es.mathworks.com/help/matlab/ref/dir.html

[27] https://es.mathworks.com/help/matlab/ref/audioread.html?searchHighlight=audioread&s_tid=doc_srchtitle

[28] http://kenschutte.com/midi

[29] Diederik P. Kingma, Jimmy Lei Ba (2017, January 30) *ADAM: A Method for Stochastic Optimization* https://arxiv.org/pdf/1412.6980.pdf

[30] https://es.mathworks.com/help/matlab/ref/audiowrite.html?s_tid=doc_ta

[31] https://es.mathworks.com/help/matlab/ref/fprintf.html

[32] https://es.mathworks.com/help/matlab/ref/fopen.html?searchHighlight=fileID&s_tid=doc_srchtitle

[33] Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron Courville, Yoshua Bengio (2017, February 11) *SampleRNN: An Unconditional End-To-End Neural Audio Generation Model* https://arxiv.org/pdf/1612.07837.pdf

[34] *What are gated recurrent units and how can they be implemented using TensorFlow?* https://www.quora.com/What-are-gated-recurrent-units-and-how-can-they-be-implemented-using-TensorFlow

[35] Wikipedia (2017, May 21) *Gated Recurrent Unit* https://en.wikipedia.org/wiki/Gated_recurrent_unit

[36] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio (2014, September 3) *Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation* https://arxiv.org/pdf/1406.1078v3.pdf

[37] Wikipedia (2017, October 3) Long Short Term Memory https://en.wikipedia.org/wiki/Long_short-term_memory

[38] *A Beginner's Guide to Recurrent Networks and LSTM's* https://deeplearning4j.org/lstm.html

[39] (2015, August 27) *Understanding LSTM Networks* http://colah.github.io/posts/2015-08-Understanding-LSTMs/

[40] http://pythonhosted.org/natsort/natsorted.html

[41] Albert Aparicio Isarn (2017, May 15) *Voice Conversion using Deep Learning* https://upcommons.upc.edu/bitstream/handle/2117/105638/AparicioAlbert_FinalReport.pdf?sequence=1&isAllowed=y

[42] Sebastian Ruder (2016, January 16) *An Overview of Gradient Descent Optimization Algoritms* http://ruder.io/optimizing-gradient-descent/index.html#adam

ANNEX1: SCRIPT FOR AUDIO REGULARIZATION

```
%Script to resize, quantize and downsample the audio files.

clear all
listroot = dir('Samples_44100_16bits'); %get the directories inside the folder
cd Samples_44100_16bits %go to the folder
fs = 44100; %sampling frequency
fsout = 16000; %output sampling frequency
warning('off','all') %turn off warnings
count = 0;
fo=[ones(48000,1);(linspace(1,0,16000))']; %envelope controller (linear volume reducion in the final 16000 samples). fade-out
%fo=[ones(12000,1);(linspace(1,0,4000))'];

for ix=4:length(listroot) %iteration for the folders in the directory
    listin = dir(listroot(ix).name); %get the directories in the ix position of the directory list
    cd(listroot(ix).name) %go to that folder
    for iy = 4:length(listin) %iteration for the files in the directory
        xin = audioread(listin(iy).name); %import the iy audio file
        x = xin/max(abs(xin)); %normalization

        [st,en] = findlim(x,0.008,fs); %function that returns the index at which the signal is the 0.8% of the max.
        if (en+400)<length(x) %controlling index exceds matrix dimentions
```

```
            x2 = x((st-100):(en+400)); %increasing the span in 500 samples to avoid harsh
variations
        x = [zeros(fs,1);x2]; %zero padding of 1second
    else
        x2 = x((st-100):end); %increasing the span in 100 samples to avoid harsh variations
        x = [zeros(fs,1);x2]; %zero padding of 1second
    end

    if length(x)>4*fs %controlling index exceds matrix dimentions
        x = x(1:4*fs); %truncating the audio file to 4seconds
    else
        x = [x ; zeros((4*fs-length(x)),1)]; %truncating the audio file to 4seconds
    end
    x = srconv(x,fs,fsout); %downsampling
    %obtention of the name of the new file and quantization to 8 bit
    %signal
    name = strsplit(listin(iy).name,'.');
    sprintf('%s',char(name(1)))
    audiowrite(strcat(char(name(1)),'_mod.wav'),x.*fo,fs,'BitsPerSample',16);
        %audiowrite(strcat(char(name(1)),'_trim.wav'),x(16000:31999).*fo,fs,'BitsPerSample',
16);
    count = count + 1;
  end
  cd ..
end
cd ..
sprintf('%s : %i elements','Finished',count)
```

```
function [ st , en ] = findlim( x , th )

%function to find the start and the end of the audio signal given a desired
%threshold
%Inputs:
%  x: input signal
%  th: threshold
% Output:
%   st: start index of the signal
%   en: end index of the signal
tmp=find(abs(hilbert(x))>th); %tmp vector containing the vector positions where the amplitude is greater than the 0.8% of the max (as it is normalised)
%first and last position of the vector gives the start and end of the audio
%signal (approximate)
st=tmp(1);
en=tmp(end);
end

function [y] = srconv(x,fsin,fsout)
% function to convert sampling rate from one sampling rate to another
% so long as the sampling rates have an integer least common multiple
% Inputs:
%   x: input signal at rate fsin
%   fsin: sampling rate on input
%   fsout: new sampling rate on output
% Output:
```

```
%   y: output signal at sampling rate fsout
% determine m, the least common multiple (lcm) of fsin and fsout
    m=lcm(fsin,fsout);
% determine the up and down sampling rates
    up=m/fsin;
    down=m/fsout;
% resample the input using the computed up/down rates
    y=resample(x,up,down);
end
```

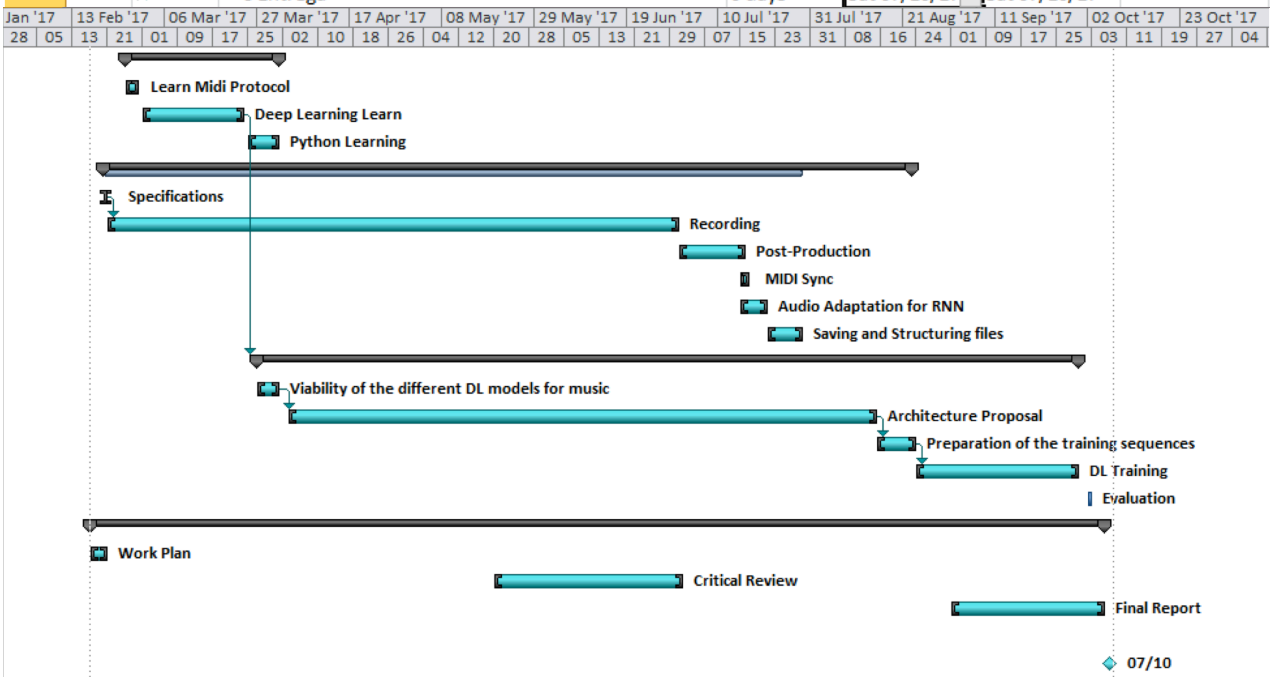
ANNEX2: SCRIPT FOR GENERATING TXT

```
clear all

listroot = dir('Samples_16000_8bits_nl'); %get the directories inside the folder

cd Samples_16000_8bits_nl %go to the folder

fs = 16000; %output sampling frequency

warning('off','all') %turn off warnings

count = 0;

for ix=4:2:length(listroot) %iteration for the folders, the step is 2 to bypass midi files with the same name as the folders

    namenote = {'A.mid'; 'Bb.mid'; 'C.mid'; 'F.mid'; 'G.mid'}; %name of the midi file

    midi = readmidi(char(namenote(0.5*ix-1))); %import midi

        [notes,endtime] = midiInfo(midi); %returns a matrix notes that contains the notes messages in a matrix

    note = notes(2,3); %gets the value of the note

    arr = [zeros(fs,1) ; note.*ones(1.5*fs,1) ; zeros(1.5*fs,1)]; %creation of the note array

    listin = dir(listroot(ix).name); %get the directories in the ix position of the directory list

    cd(listroot(ix).name) %go to that folder

    for iy = 3:length(listin) %iteration for the files in the directory

        x = audioread(listin(iy).name); %import the iy audio file

            [st,en] = findlim(x,0.008); %function that returns the index at which the signal is the 0.8% of the max

        if en+400<length(x) %controlling index exceds matrix dimentions

            en = en + 400; %tail preservation by adding 400 samples

        end

        if st-100>0

            st = st - 100; %transient preservation by adding 100 samples

        end
```

```
        out=[zeros(st,1) ; (en-st-1:-1:0)' ; zeros((length(x)-en),1)]; %forward seeing vector
        %creation of the txt file
        tmp2 = strsplit(listin(iy).name,'.');
        tmp2 = strcat(tmp2(1),'.txt');
        fileID = fopen(char(tmp2),'w');
        mat=[out';arr'];
        fprintf(fileID,'%5.0f\t%5.0f\n',mat);
        fclose(fileID);
        count = count + 1;
    end
    cd ..
end
cd ..
sprintf('%s : %i elements','Finished',count)
```

ANNEX3: GANTT DIAGRAM AND TIME PLAN

	Task Mode	Task Name	Duration	Start	Finish	Predecessors
1		**1 Preparation and Learning Process**	**25 days**	**Sat 25/02/17**	**Fri 31/03/17**	
2		1.1 Learn Midi Protocol	2 days	Sat 25/02/17	Mon 27/02/17	
3		1.2 Deep Learning Learn	17 days	Wed 01/03/17	Thu 23/03/17	
4		1.3 Python Learning	6 days	Sat 25/03/17	Fri 31/03/17	
5		**2 Sound Base Production**	**132 days**	**Mon 20/02/17**	**Tue 22/08/17**	
6		2.1 Specifications	1 day	Mon 20/02/17	Mon 20/02/17	
7		2.2 Recording	94 days	Tue 21/02/17	Fri 30/06/17	6
8		2.3 Post-Production	12 days	Sat 01/07/17	Sat 15/07/17	
9		2.4 MIDI Sync	2 days	Sat 15/07/17	Sun 16/07/17	
10		2.5 Audio Adaptation for RNN	5 days	Sat 15/07/17	Thu 20/07/17	
11		2.6 Saving and Structuring files	6 days	Fri 21/07/17	Fri 28/07/17	
12		**3 Deep Learning Model**	**135 days**	**Mon 27/03/17**	**Fri 29/09/17**	**3**
13		3.1 Viability of the different DL models for music	5 days	Mon 27/03/17	Fri 31/03/17	
14		3.2 Architecture Proposal	96 days	Mon 03/04/17	Mon 14/08/17	13
15		3.3 Preparation of the training sequences	7 days	Tue 15/08/17	Wed 23/08/17	14
16		3.4 DL Training	27 days	Thu 24/08/17	Fri 29/09/17	15
17		4 Evaluation	1 day	Mon 02/10/17	Mon 02/10/17	
18		**5 Documentation**	**166 days**	**Fri 17/02/17**	**Thu 05/10/17**	
19		5.1 Work Plan	2 days	Fri 17/02/17	Mon 20/02/17	
20		5.2 Critical Review	32 days	Sat 20/05/17	Sat 01/07/17	
21		5.3 Final Report	26 days	Fri 01/09/17	Thu 05/10/17	
22						
23		6 Entrega	0 days	Sat 07/10/17	Sat 07/10/17	



ANNEX4: BASH OUTPUT FOR CODE MODIFICATION

In this annex the loss values for the training of each batch. In the step between epochs, the model is stored and audio samples are generated which is the part that returns the error and it is in need of a revision.

(/Users/vbadenas/miniconda3) MacBook-Pro-de-Victor:SampleRNNcond vbadenas$ python train.py --exp PRUEBA --frame_sizes 16 4 --n_rnn 2 --epoch_limit 2 --dataset datasetVBC

training_loss: 0.0000 (0.0000) time: 0s

training_loss: 0.0000 (0.0000) time: 58s

training_loss: 0.0000 (0.0000) time: 112s

training_loss: 0.0000 (0.0000) time: 962s

training_loss: 0.0000 (0.0000) time: 1018s

training_loss: 0.0000 (0.0000) time: 1079s

training_loss: 0.0000 (0.0000) time: 1141s

training_loss: 0.0000 (0.0000) time: 1207s

training_loss: 0.0000 (0.0000) time: 1280s

training_loss: 0.0000 (0.0000) time: 1362s

training_loss: 0.0000 (0.0000) time: 1461s

training_loss: 0.0000 (0.0000) time: 1571s

training_loss: 0.0000 (0.0000) time: 1685s

training_loss: 0.0000 (0.0000) time: 1804s

training_loss: 0.0000 (0.0000) time: 1927s

training_loss: 1.1093 (0.0111) time: 2015s

training_loss: 3.0088 (0.0411) time: 2071s

training_loss: 2.9893 (0.0706) time: 2129s

training_loss: 2.8245 (0.0981) time: 2189s

training_loss: 2.8122 (0.1252) time: 2242s

training_loss: 2.8055 (0.1520) time: 2299s

training_loss: 2.7814 (0.1783) time: 2353s

training_loss: 2.7691 (0.2042) time: 2406s

training_loss: 2.7605 (0.2298) time: 2460s

training_loss: 2.7677 (0.2552) time: 2513s

training_loss: 2.7706 (0.2803) time: 2570s

training_loss: 2.6931 (0.3045) time: 2626s

training_loss: 2.6640 (0.3281) time: 2682s

training_loss: 2.6129 (0.3509) time: 2736s

training_loss: 2.5790 (0.3732) time: 2790s

training_loss: 2.5510 (0.3950) time: 2846s

training_loss: 2.5238 (0.4163) time: 2900s

training_loss: 2.5175 (0.4373) time: 2952s

training_loss: 2.4929 (0.4578) time: 3006s

training_loss: 2.4704 (0.4779) time: 3059s

training_loss: 2.4625 (0.4978) time: 3113s

training_loss: 2.4243 (0.5171) time: 3165s

training_loss: 2.3112 (0.5350) time: 3217s

training_loss: 2.0943 (0.5506) time: 3269s

training_loss: 1.7596 (0.5627) time: 3322s

training_loss: 1.4101 (0.5712) time: 3374s

training_loss: 1.0891 (0.5763) time: 3426s

training_loss: 0.8536 (0.5791) time: 3479s

training_loss: 0.6651 (0.5800) time: 3530s

training_loss: 0.5192 (0.5794) time: 3581s

training_loss: 0.4033 (0.5776) time: 3634s

training_loss: 0.2976 (0.5748) time: 3687s

training_loss: 0.2272 (0.5713) time: 3739s

training_loss: 0.1704 (0.5673) time: 3791s

training_loss: 0.1094 (0.5627) time: 3842s

training_loss: 0.0662 (0.5578) time: 3894s

training_loss: 0.0354 (0.5525) time: 3946s

training_loss: 0.0136 (0.5472) time: 3999s

training_loss: 0.0087 (0.5418) time: 4052s

training_loss: 0.0065 (0.5364) time: 4105s

training_loss: 0.0083 (0.5311) time: 4159s

training_loss: 0.0078 (0.5259) time: 4212s

training_loss: 0.0084 (0.5207) time: 4266s

training_loss: 0.0078 (0.5156) time: 4318s

training_loss: 0.0081 (0.5105) time: 4371s

training_loss: 0.0081 (0.5055) time: 4425s

training_loss: 0.0078 (0.5005) time: 4479s

training_loss: 0.0109 (0.4956) time: 4506s

RuntimeError: invalid argument 2: dimension 2 out of range of 2D tensor at /Users/soumith/miniconda2/conda-bld/pytorch_1503975723910/work/torch/lib/TH/generic/THTensor.c:24

ANNEX5: MODIFIED CODE IN SAMPLERNN

dataset.py

```
import utils
import numpy as np
import torch
from torch.utils.data import (
    Dataset, DataLoader as DataLoaderBase
)
from librosa.core import load
from natsort import natsorted
from os import listdir
from os.path import join
class FolderDataset(Dataset):
    def __init__(self, path, overlap_len, q_levels, ratio_min=0, ratio_max=1):
        super().__init__()
        self.overlap_len = overlap_len
        self.q_levels = q_levels
        file_names = natsorted(
            [join(path + '/wav', file_name) for file_name in listdir(path + '/wav')] #'/wav' añadido
        )
        self.file_names = file_names[
            int(ratio_min * len(file_names)): int(ratio_max * len(file_names))
        ]
        # añadido por mi
        txt_names = natsorted(
            [join(path + '/txt', file_name) for file_name in listdir(path + '/txt')]
        )
        self.txt_names = txt_names[
```

```
            int(ratio_min * len(txt_names)): int(ratio_max * len(txt_names))
        ]
        assert len(file_names) == len(txt_names), 'txt and wav folders do not have the same items'

    def __getitem__(self, index):
        (seq, _) = load(self.file_names[index], sr=None, mono=True)
        data = np.loadtxt(self.txt_names[index])
        [data1, data2] = torch.chunk(torch.from_numpy(data), 2, 1)
        ret1 = torch.cat([torch.LongTensor(self.overlap_len).fill_(utils.q_zero(self.q_levels)),
                          utils.linear_quantize(torch.from_numpy(seq), self.q_levels)])
        ret2 = torch.cat([torch.LongTensor(self.overlap_len).fill_(utils.q_zero(self.q_levels)), data1.long()])
        ret3 = torch.cat([torch.LongTensor(self.overlap_len).fill_(utils.q_zero(self.q_levels)), data2.long()])
        ret = torch.squeeze(torch.stack([ret1, ret2, ret3], dim=1))
        return ret
    def __len__(self):
        return len(self.file_names)

class DataLoader(DataLoaderBase):
    def __init__(self, dataset, batch_size, seq_len, overlap_len,
                 *args, **kwargs):
        super().__init__(dataset, batch_size, *args, **kwargs)
        self.seq_len = seq_len
        self.overlap_len = overlap_len
    def __iter__(self):
        for batch in super().__iter__():
            (batch_size, n_samples, n_features) = batch.size()
            reset = True
            for seq_begin in range(self.overlap_len, n_samples, self.seq_len):
                from_index = seq_begin - self.overlap_len
                to_index = seq_begin + self.seq_len
```

```
            sequences = batch[:, from_index : to_index, :]
            input_sequences = sequences[:, : -1, :]
            target_sequences = sequences[:, self.overlap_len :, :]
            yield (input_sequences, reset, target_sequences)
            reset = False
    def __len__(self):
        raise NotImplementedError()
```

Model.py

```
import nn
import utils
import torch
from torch.nn import functional as F
from torch.nn import init
import numpy as np
from natsort import natsorted
from os import listdir
from os.path import join
import os


class SampleRNN(torch.nn.Module):
    def __init__(self, frame_sizes, n_rnn, dim, learn_h0, q_levels, path):
        super().__init__()
        self.dim = dim
        self.q_levels = q_levels
        self.path = path
        ns_frame_samples = map(int, np.cumprod(frame_sizes))
```

```
        self.frame_level_rnns = torch.nn.ModuleList([
            FrameLevelRNN(frame_size, n_frame_samples, n_rnn, dim, learn_h0)
            for (frame_size, n_frame_samples) in zip(frame_sizes, ns_frame_samples)
        ])
        self.sample_level_mlp = SampleLevelMLP(frame_sizes[0], dim, q_levels)
    @property
    def lookback(self):
        return self.frame_level_rnns[-1].n_frame_samples


class FrameLevelRNN(torch.nn.Module):
    def __init__(self, frame_size, n_frame_samples, n_rnn, dim,learn_h0):
        super().__init__()
        self.frame_size = frame_size
        self.n_frame_samples = n_frame_samples
        self.dim = dim
        h0 = torch.zeros(n_rnn, dim)
        if learn_h0:
            self.h0 = torch.nn.Parameter(h0)
        else:
            self.register_buffer('h0', torch.autograd.Variable(h0)) #


        self.input_expand = torch.nn.Conv1d(
            in_channels=3*n_frame_samples,
            out_channels=dim,
            kernel_size=1
        )
        init.kaiming_uniform(self.input_expand.weight)
```

```
init.constant(self.input_expand.bias, 0)
self.rnn = torch.nn.GRU(
    input_size=dim,
    hidden_size=dim,
    num_layers=n_rnn,
    batch_first=True
)
for i in range(n_rnn):
    nn.concat_init(
        getattr(self.rnn, 'weight_ih_l{}'.format(i)),
        [nn.lecun_uniform, nn.lecun_uniform, nn.lecun_uniform]
    )
    init.constant(getattr(self.rnn, 'bias_ih_l{}'.format(i)), 0)
    nn.concat_init(
        getattr(self.rnn, 'weight_hh_l{}'.format(i)),
        [nn.lecun_uniform, nn.lecun_uniform, init.orthogonal]
    )
    init.constant(getattr(self.rnn, 'bias_hh_l{}'.format(i)), 0)
self.upsampling = nn.LearnedUpsampling1d(
    in_channels=dim,
    out_channels=dim,
    kernel_size=frame_size
)
init.uniform(
    self.upsampling.conv_t.weight, -np.sqrt(6 / dim), np.sqrt(6 / dim)
)
init.constant(self.upsampling.bias, 0)
```

```
    def forward(self, prev_samples, upper_tier_conditioning, hidden):
        (batch_size, _, _) = prev_samples.size()
        input = self.input_expand(
          prev_samples.permute(0, 2, 1)
        ).permute(0, 2, 1)
        #print(input.size())
        if upper_tier_conditioning is not None:
            input += upper_tier_conditioning
        reset = hidden is None
        if hidden is None:
            (n_rnn, _) = self.h0.size()
            hidden = self.h0.unsqueeze(1) \
                           .expand(n_rnn, batch_size, self.dim) \
                           .contiguous()
        (output, hidden) = self.rnn(input, hidden)
        output = self.upsampling(
            output.permute(0, 2, 1)
        ).permute(0, 2, 1)
        return (output, hidden)


class SampleLevelMLP(torch.nn.Module):
    def __init__(self, frame_size, dim, q_levels):
        super().__init__()
        self.q_levels = q_levels
        self.embedding = torch.nn.Embedding(
            self.q_levels,
            self.q_levels
```

```
        )
        self.input = torch.nn.Conv1d(
            in_channels=q_levels,
            out_channels=dim,
            kernel_size=frame_size,
            bias=False
        )
        init.kaiming_uniform(self.input.weight)

        self.hidden = torch.nn.Conv1d(
            in_channels=dim,
            out_channels=dim,
            kernel_size=1
        )
        init.kaiming_uniform(self.hidden.weight)
        init.constant(self.hidden.bias, 0)
        self.output = torch.nn.Conv1d(
            in_channels=dim,
            out_channels=q_levels,
            kernel_size=1
        )
        nn.lecun_uniform(self.output.weight)
        init.constant(self.output.bias, 0)
    def forward(self, prev_samples, upper_tier_conditioning):
        (batch_size, _, _) = upper_tier_conditioning.size()
        prev_samples = self.embedding(
            prev_samples.contiguous().view(-1)
```

```
        ).view(
            batch_size, -1, self.q_levels
        )
        prev_samples = prev_samples.permute(0, 2, 1)
        upper_tier_conditioning = upper_tier_conditioning.permute(0, 2, 1)

        x = F.relu(self.input(prev_samples) + upper_tier_conditioning)
        x = F.relu(self.hidden(x))
        x = self.output(x).permute(0, 2, 1).contiguous()
        return F.log_softmax(x.view(-1, self.q_levels)) \
                .view(batch_size, -1, self.q_levels)

class Runner:
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.reset_hidden_states()
    def reset_hidden_states(self):
        self.hidden_states = {rnn: None for rnn in self.model.frame_level_rnns}
    def run_rnn(self, rnn, prev_samples, upper_tier_conditioning):
        (output, new_hidden) = rnn(
            prev_samples, upper_tier_conditioning, self.hidden_states[rnn]
        )
        self.hidden_states[rnn] = new_hidden.detach()
        return output

class Predictor(Runner, torch.nn.Module):
```

```
def __init__(self, model):
    super().__init__(model)
def forward(self, input_sequences, reset):
    if reset:
        self.reset_hidden_states()
    (batch_size, _, _) = input_sequences.size()
    (input_sequences, other1, other2) = torch.chunk(input_sequences, chunks=3, dim=2)
    upper_tier_conditioning = None
    for rnn in reversed(self.model.frame_level_rnns):
        from_index = self.model.lookback - rnn.n_frame_samples
        to_index = -rnn.n_frame_samples + 1
        prev_samples = 2 * utils.linear_dequantize(
            input_sequences[:, from_index : to_index],
            self.model.q_levels
        )
        #print(other1)
        prev_other1 = other1[:, from_index : to_index];
        prev_other2 = other2[:, from_index: to_index];

        prev_samples = prev_samples.contiguous().view(
            batch_size, -1, rnn.n_frame_samples
        )

        prev_other1 = prev_other1.contiguous().view(
            batch_size, -1, rnn.n_frame_samples
        )
```

```
            prev_other2 = prev_other2.contiguous().view(
                batch_size, -1, rnn.n_frame_samples
            )
            prev_samples = torch.cat((prev_samples.long(), prev_other1, prev_other2), dim=2)
            prev_samples = prev_samples.float()
            upper_tier_conditioning = self.run_rnn(
                rnn, prev_samples, upper_tier_conditioning
            )
        bottom_frame_size = self.model.frame_level_rnns[0].frame_size
        mlp_input_sequences = input_sequences \
            [:, self.model.lookback - bottom_frame_size :]
            return self.model.sample_level_mlp(
            mlp_input_sequences, upper_tier_conditioning
        )


class Generator(Runner):
    def __init__(self, model, cuda=False):
        super().__init__(model)
        self.cuda = cuda
    def __call__(self, n_seqs, seq_len):
        # generation doesn't work with CUDNN for some reason
        torch.backends.cudnn.enabled = False
        self.reset_hidden_states()
        bottom_frame_size = self.model.frame_level_rnns[0].n_frame_samples
        sequences = torch.LongTensor(n_seqs, self.model.lookback + seq_len) \
                        .fill_(utils.q_zero(self.model.q_levels))
        frame_level_outputs = [None for _ in self.model.frame_level_rnns]
```

```
path = os.path.join('datasets', 'datasetVBC')
txt_names = natsorted(
    [join(path + '/txt', file_name) for file_name in listdir(path + '/txt')]
)
data = np.loadtxt(txt_names[0])
[data1, data2] = torch.chunk(torch.from_numpy(data), 2, 1)
other1 = torch.cat([torch.LongTensor(64).fill_(utils.q_zero(128)), data1.long()])
other2 = torch.cat([torch.LongTensor(64).fill_(utils.q_zero(128)), data2.long()])
for i in range(self.model.lookback, self.model.lookback + seq_len):
    for (tier_index, rnn) in \
            reversed(list(enumerate(self.model.frame_level_rnns))):
        if i % rnn.n_frame_samples != 0:
            continue
        prev_samples = torch.autograd.Variable(
            2 * utils.linear_dequantize(
                sequences[:, i - rnn.n_frame_samples : i],
                self.model.q_levels
            ).unsqueeze(1),
            volatile=True
        )
        prev_other1 = torch.autograd.Variable(other1[:, i - rnn.n_frame_samples: i],
            volatile=True
        )
        prev_other2 = torch.autograd.Variable(other2[:, i - rnn.n_frame_samples: i],
            volatile=True
        )
        if self.cuda:
```

```
            prev_samples = prev_samples.cuda()
            prev_other1 = prev_other1.cuda()
            prev_other2 = prev_other2.cuda()
        if tier_index == len(self.model.frame_level_rnns) - 1:
            upper_tier_conditioning = None
        else:
            frame_index = (i // rnn.n_frame_samples) % \
                self.model.frame_level_rnns[tier_index + 1].frame_size
            upper_tier_conditioning = \
                frame_level_outputs[tier_index + 1][:, frame_index, :] \
                                .unsqueeze(1)

        prev_samples = torch.cat((prev_samples.long(), prev_other1, prev_other2), dim=2)
        prev_samples = prev_samples.float()
        frame_level_outputs[tier_index] = self.run_rnn(
            rnn, prev_samples, upper_tier_conditioning
        )
    prev_samples = torch.autograd.Variable(
        sequences[:, i - bottom_frame_size : i],
        volatile=True
    )
    if self.cuda:
        prev_samples = prev_samples.cuda()
    upper_tier_conditioning = \
        frame_level_outputs[0][:, i % bottom_frame_size, :] \
                        .unsqueeze(1)
    sample_dist = self.model.sample_level_mlp(
```

```
                prev_samples, upper_tier_conditioning
            ).squeeze(1).exp_().data
            sequences[:, i] = sample_dist.multinomial(1).squeeze(1)
        torch.backends.cudnn.enabled = True
        return sequences[:, self.model.lookback :]
```

__init__.py in trainer module

```
import torch
from torch.autograd import Variable
import heapq
# Based on torch.utils.trainer.Trainer code.
class Trainer(object):
    def __init__(self, model, criterion, optimizer, dataset, cuda=False):
        self.model = model
        self.criterion = criterion
        self.optimizer = optimizer
        self.dataset = dataset
        self.cuda = cuda
        self.iterations = 0
        self.epochs = 0
        self.stats = {}
        self.plugin_queues = {
            'iteration': [],
            'epoch': [],
            'batch': [],
            'update': [],
        }
    def register_plugin(self, plugin):
```

```
        plugin.register(self)
        intervals = plugin.trigger_interval
        if not isinstance(intervals, list):
            intervals = [intervals]
        for (duration, unit) in intervals:
            queue = self.plugin_queues[unit]
            queue.append((duration, len(queue), plugin))
    def call_plugins(self, queue_name, time, *args):
        args = (time,) + args
        queue = self.plugin_queues[queue_name]
        if len(queue) == 0:
            return
        while queue[0][0] <= time:
            plugin = queue[0][2]
            getattr(plugin, queue_name)(*args)
            for trigger in plugin.trigger_interval:
                if trigger[1] == queue_name:
                    interval = trigger[0]
            new_item = (time + interval, queue[0][1], plugin)
            heapq.heappushpop(queue, new_item)
    def run(self, epochs=1):
        for q in self.plugin_queues.values():
            heapq.heapify(q)
        for self.epochs in range(self.epochs + 1, self.epochs + epochs + 1):
            self.train()
            self.call_plugins('epoch', self.epochs)
    def train(self):
```

```
for (self.iterations, data) in \
        enumerate(self.dataset, self.iterations + 1):
    batch_inputs = data[: -1]
    batch_target = data[-1]
    self.call_plugins(
        'batch', self.iterations, batch_inputs, batch_target
    )
    def wrap(input):
        if torch.is_tensor(input):
            input = Variable(input)
            if self.cuda:
                input = input.cuda()
        return input
    batch_inputs = list(map(wrap, batch_inputs))
    batch_target = Variable(batch_target)
    #print(batch_inputs)
    (batch_target, _, _) = torch.chunk(batch_target, chunks=3, dim=2)
    batch_target = torch.squeeze(batch_target, dim=-1)
    if self.cuda:
        batch_target = batch_target.cuda()
    plugin_data = [None, None]
    def closure():
        batch_output = self.model(*batch_inputs)
        #print(batch_output)
        loss = self.criterion(batch_output, batch_target)
        loss.backward()
        if plugin_data[0] is None:
```

```
            plugin_data[0] = batch_output.data
            plugin_data[1] = loss.data
        return loss
    self.optimizer.zero_grad()
    self.optimizer.step(closure)
    self.call_plugins(
        'iteration', self.iterations, batch_inputs, batch_target,
        *plugin_data
    )
    self.call_plugins('update', self.iterations, self.model)
```